The Journal of Supercomputing
https://doi.org/10.1007/s11227-018-2682-1

PSML: parallel system modeling and simulation language for electronic system level

Alireza Poshtkohi[1] · M. B. Ghaznavi-Ghoushchi[1] · Kamyar Saghafi[1]

Abstract

System-level description languages in electronic system level are domain-specific sequential simulation languages using the shared-everything model in discrete-event modeling and simulation terminology. They implement sequential process-interaction worldview to take advantage of event partitioning for ease of programming and modularity. Therefore, their reference simulators can only be executed on a single physical core. Fast and accurate simulation is highly desirable for efficient and effective system design due to the ever-increasing complexity of embedded and cyber physical systems. Parallel discrete-event simulation (PDES) is the main technique to solve this problem for large-scale system-level models. PDES works based on state space partitioning by using the so-called logical process worldview. This paper proposes parallel system modeling and simulation language (PSML), along with its formalized distributed parallel simulation kernel, that provides execution of hardware models in order to improve simulation speed significantly. It will be shown that the proposed framework results in linear, super-linear speedups ranging from 11 $\times$ to 32 $\times$ for large-scale, complex PSML models in comparison with the SystemC reference simulator on a 12-core host.

Keywords Parallel discrete-event simulation (PDES) · Parallel simulation languages (PSLs) · System-level description languages (SLDLs) · SystemC · Transaction-level modeling (TLM) · Embedded systems

✉ M. B. Ghaznavi-Ghoushchi
ghaznavi@shahed.ac.ir

Alireza Poshtkohi
arp@poshtkohi.ir

Kamyar Saghafi
saghafi@shahed.ac.ir

[1] Department of Electrical Engineering, Shahed University, Tehran 3319118651, Iran

Published online: 12 November 2018

1 Introduction

The large size and complexity of the modern digital hardware impose great challenges to design and validation. Today, computer simulation is a well-accepted technique to verify Hardware (HW)/Software (SW) embedded systems. System-level design addresses the complexity issue of embedded systems by raising the level of abstraction [1]. This complexity imposes significant challenges for modeling, verification and synthesis in digital hardware design. Moving to higher abstraction levels through system-level languages allows designers to describe hardware and software components based on the same level or mixed levels. Hardware models are usually written in C/C++-based System-Level Description Languages (SLDLs). According to the trends in integrating more cores into a single chip and the steady growth in system size and complexity, simulation time has greatly increased and so costs and verification time have risen. The main technique to solve this problem is parallel and distributed simulation (PADS) [2–5]. SLDLs rely on *sequential discrete-event simulation* (DES) and are of *sequential simulation languages* using the *shared-everything* model [6]. They implement *sequential process-interaction worldview* to take advantage of *event partitioning* for ease of programming and modularity.

PDES exploits the natural parallelism in simulation models to improve simulation speed substantially. The advent of multi-core and many-core architectures provides new opportunities to revisit fine-grained applications like PDES. There are different sequential simulation languages (SSLs) to model digital systems. They differ in usability due to their underlying semantic models. PDES works reliant on the so-called *logical process* (LP) worldview [2]. This formalism cannot be applied directly to SLDLs because they are SSLs. There are two key features for the LP pattern that are shared by all PDES platforms: *state space partitioning* and *event scheduling*. A novel formalism is required to specify these characteristics in electronic system level (ESL). PDES is entirely and fundamentally different from sequential DES, and thus, a domain-specific PDES-compliant simulation language for HW systems must be developed with its own theory and life cycle largely independent of DES.

In summary, the key contributions of this paper are as follows:

- It proposes parallel system modeling and simulation language (PSML) for ESL. This language can be directly used as either a new parallel SLDL to model digital hardware or an intermediate language in order to map other SLDLs by using common compiler techniques. This avoids repetition of basic discoveries when parallelizing the existing SLDLs and Hardware Description Languages (HDLs).
- It formally defines PSML parallel execution semantics on a PDES abstract machine which is consistent with different PDES algorithms (conservative or optimistic).
- A proof-of-concept implementation of the PSML execution semantics is presented in a highly extensible and optimized parallel environment.
- It proposes a new timing model for the semantics of event ordering to tackle the problem of *simultaneous events* for deterministic parallel simulation of SLDLs.

The rest of the paper is organized as follows. In Sect. 2, we study some fundamental concepts, and the factors that motivated us to propose PSML language. Section 2 also focuses on the related works published so far for parallel simulation in the context of

this article. PSML language is introduced in Sect. 3. Section 4 mainly discusses PSML simulation semantics and implementation of its PDES-compliant simulation kernel. Some PSML experimental studies are described in Sect. 5. Section 6 concludes the paper and presents our future schedule to extend PSML framework.

2 Basic concepts and motivation

2.1 PDES fundamentals

In distributed simulation or PDES, the entire simulation state is divided into a collection of smaller non-overlapping states managed by subtasks referred to as LPs, and each of them is executed by a processor or node [5]. These LPs communicate with each other by exchanging timestamped event messages in simulated time. These events are used to synchronize the execution of LPs in parallel. An event refers to an update to system state at a particular instant in simulation time. LPs do not share any state variables and solely communicate through these timestamped messages. Synchronization between LPs is violated when one of the LPs receives an out-of-order event. To overcome this problem, a lot of synchronization protocols have been proposed that have mainly fallen into synchronous and asynchronous category. In synchronous simulation, all the LPs see a single clock and synchronize with one another through an expensive global barrier mechanism after processing events in current simulation time, where parallelism can only be exploited if a model contains simultaneous events. A shared memory (multi-core) synchronous implementation of a DES program does not need any knowledge of the LP worldview. In asynchronous algorithms, LPs process events at different times, namely different local clocks. Asynchronous conservative algorithms strictly avoid causality violations, such as CMB, by blocking on input channels and provide mechanisms to resolve deadlocks: deadlock avoidance by null messages and deadlock recovery by deadlock detection. Optimistic protocols allow violations and provide mechanisms to recover. Undoing modification of state variables can be accomplished by taking a snapshot of the state of each LP prior to processing each event. One of the central issues in Time Warp synchronization is how to support fast state restoration with low-overhead checkpointing [2]. Asynchronous synchronization is much more complicated than synchronous ones, but reaches much higher speed and strong scaling [7] because processors process events with different timestamp and the properties of pipeline execution are seen. Both conservative and optimistic variants assume all events in the simulation have unique timestamps and so they guarantee deterministic parallel simulation and reproducible results. Therefore, simultaneous events must be treated precisely for those applications that generate them, including HDLs and SLDLs.

2.2 Problem definition

Standardized IEEE HDLs and SLDLs—including VHDL, Verilog, SystemC, SystemVerilog and SpecC [8–12]—allow hardware designers to describe parallel components of an electronic system and concurrent interactions between them. They are

actually *domain-specific sequential simulation languages* for hardware design in terms of modeling and simulation, because sequential simulation languages are based on the process-interaction approach and routinely use the *shared-everything* model [6]. The simulation of a hardware model is full of simultaneous events that are handled by on the well-known delta cycles. This technique imposes a partial order on simultaneous events for sequential deterministic execution. These languages take advantage of co-routine semantics (suspend and resume) invented in Simula simulation language [13] by *wait* expression to provide concurrent simulation of simultaneous events. Every (simulation) language that makes use of co-routines is also called a *concurrent language*. Due to existence of simultaneous events, legacy hardware languages can be easily accelerated by synchronous parallelization approaches and protecting the shared state on shared memory machines; however, they are not efficient because of the lockstep execution and low degree of parallelism. *Parallel simulation languages* (PSLs) work reliant on the *state space partitioning* by *logical process* worldview [2, 14] for parallelism purposes while a *sequentially process-oriented simulation language* takes advantage of *event partitioning*. PDES has recently gained extensive attention for parallelizing sequential SLDLs due to the ever-increasing complexity of embedded and cyber physical systems and pervasive availability of the multi-core hosts within exascale era. In this paper, we look at the main problem from a radically different angle and propose a new parallel simulation language—or more precisely a PDES language—called PSML to model and simulate hardware systems. In PSML, the modeler is provided by an abstract view of a model of computation (MoC) for PDES that hides the underlying complex synchronization so that he can benefit from an explicit PSL to leverage maximum efficiency of the parallel execution. The defined abstractions let PSML developers extend the language and its simulation semantics with minimum knowledge of PDES. Additionally, they allow PDES kernel developers to optimize the underlying synchronization algorithms to meet the demands of PSML kernel with little expertise in SLDLs. Based on the above goals, we propose PSML conceptual framework in Fig. 1, which is aimed at the following main characteristics:

- *PSML parallel programming model* It should fully inherit SLDL features such as modular design, synchronization, flexible timing, and so forth. A key principle in PSML was to make everything explicit to the designer except for PDES synchronization. It shall support parallel programming primitives available in traditional parallel programming languages by explicit means of host parallelism, model decomposition, processor mapping and communication. Because parallelism is explicit, programmer can write an efficient program and tune it for peak performance. LPs must be able to directly address each other through ports. This helps the designer have full access to communication details of the design at runtime in order to effectively partition the model and map the LPs to processors. The control of granularity is given to user as the outcome of explicit parallelism, because it has an important impact on parallel simulation speed. Users should be able to take control of shared variables without knowledge of the principles in distributed shared memory (DSM) consistency protocols. Language features must facilitate distributed and optimistic executability.

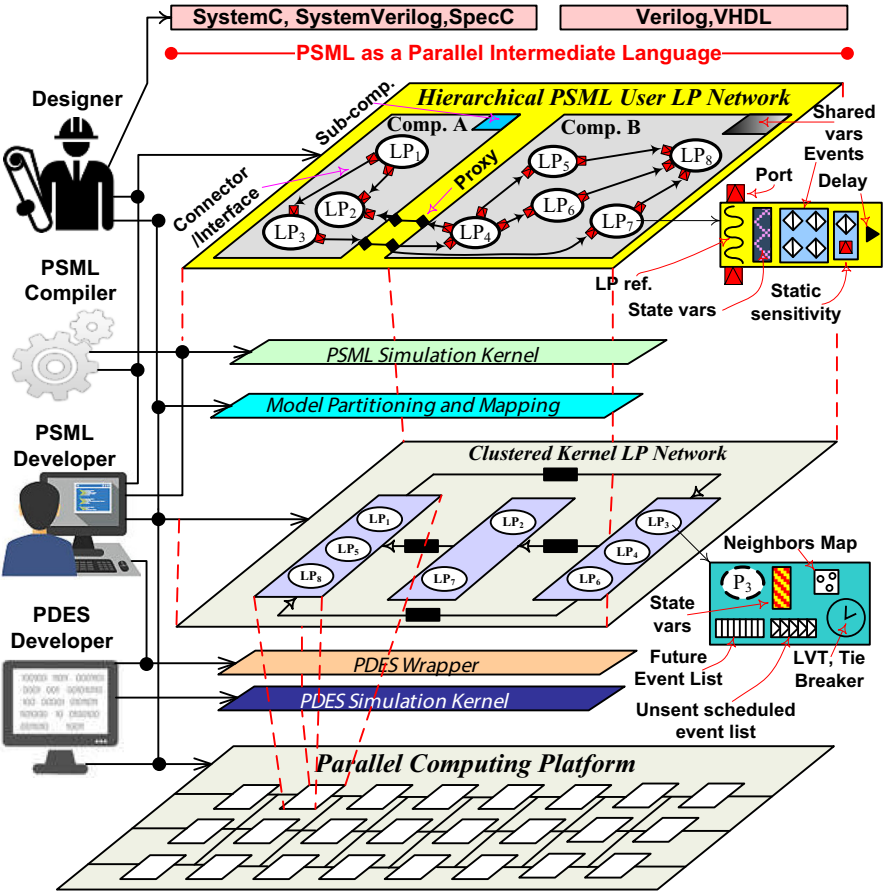

Fig. 1 PSML conceptual framework

- *PSML parallel execution semantics* Simulation execution semantics shall be formalized on a generalized PDES abstract machine that complies with the LP-based event-scheduling worldview. This implicitly states that many advantages can be brought from PDES literature to PSML. For example, the language can be easily implemented on shared memory (SM) and distributed shared memory (DSM) architectures by various PDES algorithms whether conservative or optimistic. Dynamic load-balancing techniques through process migration or adaptive PDES strategies become seamlessly straightforward. PDES algorithms are complex. To make the complexity of PSML kernel manageable, separation of concerns shall be considered. For instance, language constructs that generate PDES events should be classified, and execution semantics is defined separately for each one. Deterministic simulation must be guaranteed under execution of different PDES algorithms (such as CMB and Time Warp) by defining a concise timing model for event ordering. It should introduce possible system-level optimizations for simulation speed, including efficient communication and dynamic memory management.

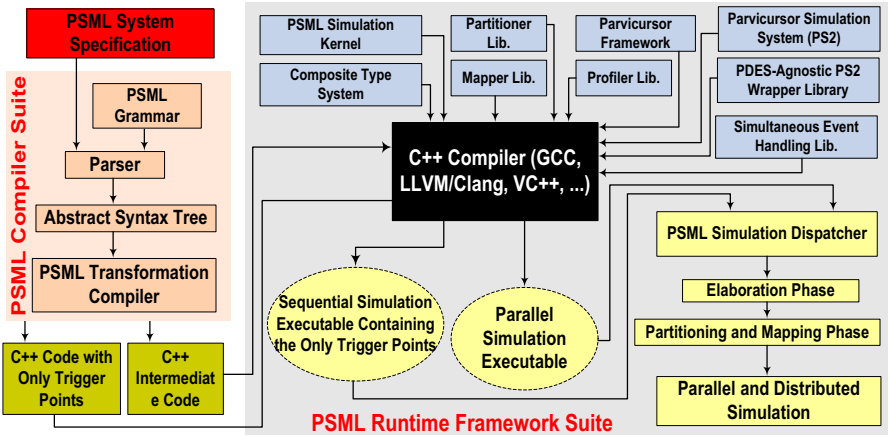

Fig. 2 The PSML tool flow: PSML compiler suite and parallel execution framework

- *PDES synchronization* Execution semantics must be implemented finally by a system-level simulation kernel. We shall determine the requirements of this kernel for how to interface with PDES layer and specify its properties. In this paper, we define a wrapper facade to address this problem. The opportunity for possible optimizations in PDES layer should be investigated. This key metric is examined from scalability's point of view in context of a new PDES simulation kernel built for PSML.

Figure 2 shows the PSML tool flow and parallel execution toolset. A PSML model is first compiled and then executed on top of the parallel computing platform. There are several tools, components and runtime libraries, which are studied in this paper.

2.3 Related work

Several formalisms and sequential languages exist to specify discrete-event systems, including HDLs, SLDLs, DEVS, Statecharts, Simula, CSIM, DFSS [15], and so on. In hardware design, PDES has predominantly been used for parallel simulation of gate-level logic circuits expressed by netlists due to its simplicity. Numerous works explore different conservative and optimistic protocols along with partitioning and load balancing algorithms at Gate Level (GL) [16–21]. PDES has also applied to a limited synthesizable subset of HDLs that are close to GL using PDES [22–24]. In the realm of SLDLs, there are some works to parallelize SpecC and SystemC while SystemVerilog has not yet been taken into consideration. Because SLDLs usually support different abstraction levels and are very complex, the majority of reported works only consider a single abstraction level for parallel simulation. Also, most works make use of synchronous PDES algorithms on multi-core systems. In other words, there are many restrictions toward them that they cannot even be used with distributed asynchronous algorithms. All these techniques directly parallelize the reference simulation kernel which may not work on new kernel releases.

In [25, 26], SpecC models are executed in parallel using an improved version of the synchronous PDES algorithm applied to the sequential simulation kernel on multi-core machines. In [27–29], SystemC kernel is modified to allow parallel simulation on multi-core and GPU systems at Register Transfer Level (RTL). Other works have been reported toward SystemC kernel parallelization for transaction-level modeling (TLM) simulation that are also implemented atop synchronous PDES [30–38], a complete survey can be found in [39]. There are also a number of distributed synchronous implementations by modifying the SystemC kernel at RTL [40, 41]. As a whole, all the fully fledged parallelization works of hardware-specific SLDLs (i.e., SystemC, SystemVerilog and SpecC) benefit from synchronous PDES approaches, because this model is inherently compatible with SSLs. Instead, asynchronous PDES is widely used in PADS community and all the PSLs had been implemented atop these two protocols.

As identified by Fujimoto [3], one of the six major research challenges in PADS is to make it widely accessible to the general Modeling and Simulation (M&S) community by simplifying the development of simulation models. Three main approaches used to address this problem are: (1) PDES kernel libraries; (2) development of automatic parallelization methods for sequential simulation languages; and (3) new parallel simulation languages. To date, progress has been made greatly in areas (1) and (2). Since programming with PDES libraries is cumbersome for modelers unaccustomed to PDES and they do not support modular design, a few of PSLs were created in the 1990s. However, their development was abandoned because they did not cover the broad requirements of underlying parallel processing and fully fledged object-oriented programming constructs available in languages like C++ and Java. If they resolved these problems, it would be greatly favorable to convert SSLs, such as SystemC, DEVS, etc., into those PSLs. For example, parallelization techniques of all the hardware languages either use a third-party PDES library or extend their kernel relied on conservative PDES. Therefore, new domain-specific modeling languages coupled with mechanisms to automatically translate modeling abstractions to efficient parallel simulation code are an important avenue for M&S community. Suitable intermediate representations are needed to provide descriptive yet precise specifications of model state and behavior [3]. PSML is an essential attempt to develop a new PDES language for ESL in this regard, of course, PSML defines a powerful programming interface that can be used to model other DES problems like large-scale networked systems [42]. PSLs were usually built by adding primitives or library functions to a sequential simulation language to specify parallel execution. However, PSML has been designed from the ground up with the aim of exposing explicit parallelism in mind.

Objected-oriented (OO) PSLs only supports a very basic part of OO concepts, in which each entity contains a set of processes communicating by exchanging messages. They are deficient in common OO concepts such as templates, virtual methods, operator overloading, polymorphism, abstract classes, etc., which are compulsory for system-level design. They need a PDES-aware compiler that is used to generate the underlying LP code atop a typical PDES library written in a language like C or C++. Consequently, the user is prohibited from low-level access to PDES engine if he is willing to make changes for performance optimization. Provided that the intermediate code is available, it is difficult to be understood and modified (because the user has

no knowledge of its execution semantics). They lack a model elaboration phase, and so we call them as *static languages*. This exposes the limitation on the models that cannot be created dynamically at runtime. This feature is essential for reusing third-party model libraries and design space exploration (e.g., an NoC to be reconfigured at runtime). The granularity of these languages is at the entity level in which an entire entity is mapped to an LP, and thus, they can dramatically reduce the degree of parallelism in hierarchical models. They hand the model-to-processor mapping over to the user (due to lack of an elaboration phase, he is made to map entities to processors manually, which is a very time-consuming task for large-scale models) or use the round-robin algorithm of the underlying PDES simulator. APOSTLE and Parsec languages extended the C language [43], while TeD was a domain-specific language for telecommunication networks and used a mixture of the two VHDL and C languages [44] (which was problematic for the user, because two different languages must be learnt and used to write structures and behaviors of the models).

The PSML language also targets the above issues directly. PSML is a fully fledged object-oriented language (with a dedicated elaboration phase) and has a PDES-aware syntax and therefore can be compiled by a PDES-unaware compiler. PSML takes advantage of a profile-driven sequential execution for partitioning purposes. The structure of a model is accessible to the modelers and PSML kernel as a *weighted PSML graph* (WPG) in elaboration phase, which is used to implement automatic and manual partitioning algorithms reliant on the weight information of graph nodes. This mapping is performed at the granularity of process level not PSML components. PSML, in addition to being a general-purpose PDES language, provides a broad spectrum of language constructs for ESL, including parallel hardware execution semantics, hardware-specific data types, hardware timing models, and interfaces. Furthermore, powerful PSML primitives allow the programmer to use the facilities present in common parallel programming languages like MPI [45, 46]—including, data marshaling for distributed execution, processor mapping, parallel execution statistics, memory coherence protocols, etc. As a whole, there should be a parallel simulation language for digital electronic systems, which supports HDL/SLDL features, rather than seeking how to parallelize existing languages separately. Such a PSL allows us to easily implement compilation tools in order to convert those languages to our PSL and avoid basic discoveries.

3 PSML language

3.1 The language

PSML design is inspired by parallel Parvicursor infrastructure [47] and the successful, objected-oriented language C# that is being widely used in the software industry. Parvicursor.NET Framework allows developers to implement C#-based.NET ECMA programs directly in native code. One of the primary goals of PSML language is to enable asynchronous, explicitly parallel system-level modeling—that is, real parallel modeling of systems above the RTL that might be implemented in hardware, software or a combination of the two. Of course, RTL and GL modeling are also possible in

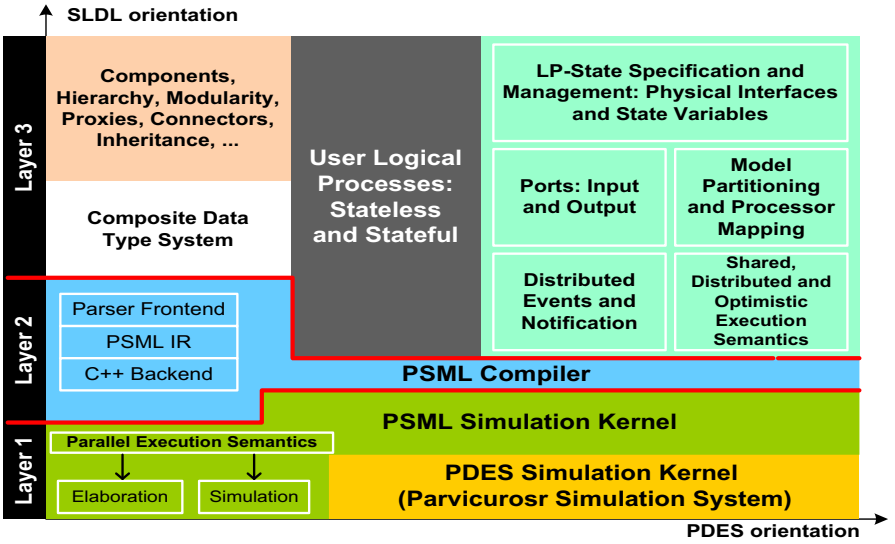

Fig. 3 PSML architecture and its orientation between an SLDL and PDES

PSML. PSML framework is completely built on top of the Parvicursor. PSML compiler emits native Parvicursor-compliant codes that can be compiled by any existing C++ compiler. Figure 3 portrays PSML architecture inside a two-dimensional coordinate system with the goal of visualizing the orientation between a SLDL and a PDES language. Those that need special attention for parallel execution are located in the direction of PDES orientation axis.

As a simple example, let's examine modeling the c17 circuit and show its structure and semantics in PSML as illustrated in Figs. 4, 5, 6 and 7. In Fig. 5, implementation begins by inheriting from the class *psml_component*. Proxies are used to connect different components to each other (lines 3 and 4). The gate LP is declared as *stateless* because it has no *explicit* state (line 14). Input and output ports are separately defined for LPs (lines 6 and 7). Here, all proxies and ports host the interface *psml_wire*.

Proxies and LPs are registered to LP ports and components, respectively, in class constructors (lines 13–14). Ports along with their sensitivity list are registered to LP handles (lines 15–16). A LP always sees its reference. In Fig. 6, the driver LP is *stateful* and *explicitly* stores its state variables in a class derived from *psml_state* class (defined in lines 2 through 5, and used in lines 13–19). A customized model partitioner and process mapper is implemented to assign the processes of a model to processors by the user (lines 14 and 15). Simulation runs on 4 cores (lines 16–19). Figure 8 presents PSML parallel execution flow. At runtime, the model is elaborated in four individual phases and transformed into a network of logical processes. A PSML model has a hierarchical, object-oriented, modular and component-based structure. Unlike all existing serial SLDLs where ports are defined just for modules, in PSML, ports are defined for processes inside a component and components are linked together through a type of connectors called proxies. In the third phase, the modeler can implement two types of fine-grained and coarse-grained partitioning algorithms for his/her design. After

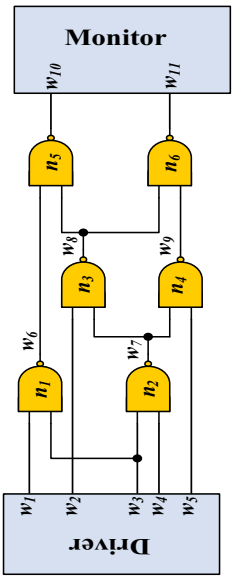

Fig. 4 The gate-level model of the c17 circuit

```
1 class nand2 : public psml_component {
2   // Proxy declarations
3   public psml_proxy<psml_wire<bool> > x0_, x1_;
4   public psml_proxy<psml_wire<bool> > y0_;
5   // Port declarations
6   private psml_inport<psml_wire<bool> > x0, x1;
7   private psml_outport<psml_wire<bool> > y0;
8   private psml_ulp lp;
9   private psml_time wire_delay(1, PSML_NS);
10  // Constructor
11  public nand2(System.String name) {
12    set_name(name);
13    register_proxy(x0, x0_); ...
14    lp = register_user_lp(ulp, false, "ulp");
15    lp.register_port(x0); ...
16    lp.register_sensitivity(x0); ...
17  }
18  // Stateless PSML User  LP
19  private void ulp(psml_ulp owner) {
20    while(true) {
21      bool Y = !(x0.fetch() & x1.fetch());
22      y0.put(Y, wire_delay, owner);
23      psml_wait(delay, owner);
24    }
25  }
26 }
```

Fig. 5 A NAND gate in PSML

partitioning, we obtain a clustered network of logical processes. Parallel simulation phase is responsible for deterministic execution of PSML models based on PSML parallel simulation semantics.

A driver component.

```
class driver : public psml_component {
  class state : public psml_state {
    public Int32 i = 0;
    public Int counter = 0;
  }
  private state s = new state();
  private psml_time wd(0, PSML_NS);
  private psml_time delay(5, PSML_NS);
  psml  vector patterns[5] = {"10001", ...};
  // Stateful PSML User LP
  private void ulp(psml_ulp owner) {
    while(true) {
       x0.put(patterns[s.i].get(0), wd, owner);
       x1.put(patterns[s.i].get(1), wd, owner);
       Console.Wrie("@time {0}",  psml_sim_time(owner));
       Console.WriteLine(" x0 x1 x2 x3 x4 {0}{1}{2}{3}{4}", x0, x1, x2, x3, x4);
       psml_wait(delay, owner);
       if(s.i ++ == 5)  s.i = 0;
       s.counter++;
    }
  }
}
```

Fig. 6 A driver component

c17 model representation.

```
using Parvicursor.psml;
using Parvicursor.PS2
void Main(System.String[] args) {
  // Sets simulation time resolution
  psml_set_time_resolution(1,  PSML_NS);
  // Instantiation
  nand2 n1("n1");
  driver d("d"); …
  // Connectivity
  psml_connector c1, c2, ...;
  psml_wire<bool> w1, w2, ...;
  n1.x0_.bind(c1, w1); …
  // Simulation
  psml_ partitioner par = new  psml_partitioner(“rb”);
  psml_mapper map = new psml_mapper(“rb”);
  UInt32 cores = 4;
  psml_simulator sim = new  psml_simulator(cores, par, map);
  psml_time t(1000,  PSML_NS);
  sim.start(t);
}
```

Fig. 7 c17 model representation

3.2 PDES-specific constructs in PSML

Designing a PSL involves balancing the aims of performance and elegance while accommodating the distributed nature of parallel simulation and the underlying synchronization mechanism. Because parallel simulations are intended to be run on distributed multiprocessor architectures, the design of the PSML must ensure the language features scale easily on them and do not rely on shared memory or centralized

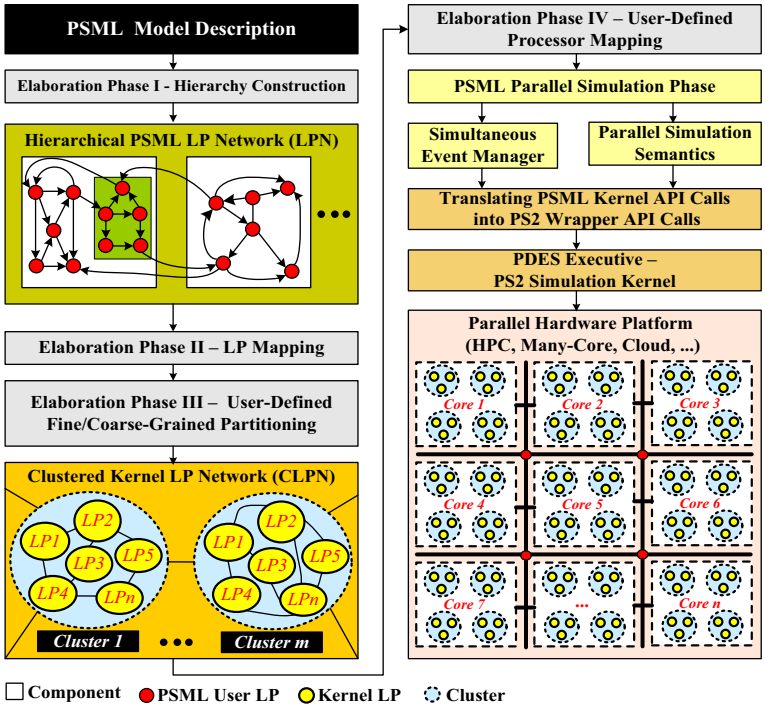

Fig. 8 Parallel PSML execution flow for simulation of SLDL models

(synchronous) synchronization. One important design issue in PSML was to choose a loosely coupled process computation model in that LPs only communicate by message passing and generally do not share read/write memory. This model is consistent with many parallel simulation protocols, including CMB and Time Warp [1]. PSML presents the user with a different clock in each LP, and each event has a timestamp and other LP-related information with it. In fact, PSML is a hierarchical, modular LP-based PSL, or more precisely an objected-oriented PDES language, that supports general-purpose discrete -event simulation semantics and domain-specific language (DSL) constructs for HW systems. Table 1 shows differences and similarities between PSML and the de-facto SLDLs.

- *Language support for PDES-aware HW specification* The atomic element of a PSML model is a *user logical process* (ULP). ULPs are connected to each other through ports and connectors. ULPs can be embedded in components and a reference to each ULP is returned to the modeler by PSML for future uses within the model, while it is not a necessity and the modeler can construct his model only by ULPs and connectors. Each component can also host a number of sub-components to build a hierarchy of components. PSML implements a concept referred to as *proxy software design pattern* to connect components and sub-components to one another and support modular design. Each ULP has a set of state variables that must be registered to the kernel. Since each ULP is sequentially executed, we suggest two simulation

Table 1 Comparison and mapping between PSML and different SLDL constructs

Feature	System-level description languages		
	PSML	SystemC	SystemVerilog
OOP styles	Fully inherited from C#	Fully implemented in C++	Classes, single inheritance, polymorphism, …
Hierarchy/modularity	*psml_component*	*SC_MODULE*	*module*
(Logical) processes	*psml_ulp ulp=register_user_lp(method,mode,name)*	*SC_METHOD/SC_THREAD*	Sequential and parallel blocks, and blocking, non-blocking and continuous assignments
Ports/proxies	*psml_inport, psml_outport, psml_connector, psml_proxy*	*sc_in, sc_out, sc_port*	*input, output*
Interfaces/channels	*psml_nominal_interface* (*psml_wire*)*: create_instance*, *destroy_instance*, *clone*, *copy*, *commit*, *delete*	*sc_signal, sc_prim_channel, sc_fifo, sc_semaphore, sc_mutex*	*wire, reg, interface*
System-level synchronization	Serializable distributed PDES events: *psml_event* (*register_initiator, register_subscriber, psml_wait, psml_notify*), *sensitivity list* (*register_sensitivity*)	*sc_event* (*wait, notify*), *sensitivity list*	*event* (*wait, trigger*), *sensitivity list* (@)
Simulation	*psml_simulator*(*start*, *stop*)	*sc_start, sc_stop, sc_pause*	*$stop, $finish*
Model partitioning/processor mapping	Weighted PSML Graph*: psml_partitioner*, *psml_mapper*	–	–
Shared memory execution semantics	Memory coherence protocols: *shared* keyword for shared variables	Cooperative Scheduling	Cooperative Scheduling
Distributed shared memory execution semantics	Serializable composite data type system (*serialize*, *deserialize*, *clone*, *copy*, *delete*), interface function calls	–	–
Optimistic execution semantics	*register_state_var,* port-bound physical interfaces	–	–
Dynamic memory allocation	*psml_pdes_memory_manager* (*psml_new, psml_delete*)	–	–

worldviews to be used by each ULP: process interaction and event scheduling. ULPs communicate with each other by exchanging explicit PDES events (which can be used by *psml_notify*, and *psml_wait*) or sensible ports. A system-level PDES event, which is a serializable object to be transferred on the network by PDES layer, has a number of registered initiators and subscribers. Notification of a PDES event by an initiator means explicit delivery of its effect to subscribers. Communication through a port is intercepted by its bound pair of *nominal interface* (NI) and *physical connector* (PC). A NI is only used for decision-making by PSML kernel when a communication is requested by a sender ULP, because two different ULPs may reside on different processors or computer nodes. A *physical interface* (PI) instance is attached to every input/output of each ULP; this is done by calling *create_instance* method, which must be implemented by the user for new interfaces, at elaboration phase.

Each interface like *psml_wire* in PSML must implement a number of methods such as *put* (to buffer a new value to PSML kernel), *fetch* (to read current value of the interface), *serialize/deserialize* (for data marshaling over a network), *copy* (to copy a new content into the data of an interface when receiving), *commit* (for intra-cluster zero-copy communication), *clone* (to duplicate from the contents of an interface for sending), *delete* (to free a memory buffer allocated by *clone*), etc.; in fact, these methods make it possible for PSML kernel to manage data transfer between ULPs' ports. Prototype of each ULP looks similar to *void my_name*(*psml_ulp owner*). Stateful ULPs are those that must explicitly define and manually manage their own state; of course, PSML transparently monitors the state of input ports for this type of ULPs to ease programming. Context switches through *psml_wait* statement is supported in both types of ULPs. PSML takes advantage of the *shared-nothing but some shared state* principle in PSLs for allowing distributed simulation and different PDES protocols such as conservative (CMB), optimistic (Time Warp) and adaptive. The facility of *some shared state*, which can be declared by the keyword *shared* in a model, only provides a basic means of shared variables for HW systems. This is necessary because PDES algorithms cannot work with the *shared-everything* coding styles used in traditional sequential simulation languages. If ULPs are using shared variables, PSML compiler emits an additional set of port/NI declarations without being registered in the sensitivity list, and the read/write calls are explicitly performed through these ports instead of the original variables. In principle, shared variables are supported by duplicating the shared information in PSML processes that need it. Because a shared variable can be modified, a protocol is required to ensure coherence among different copies of the shared state. Reads are satisfied immediately by returning the current value of the local copy while updates are broadcast to all processes through the assigned ports. Since PSML kernel makes use of a total causal event ordering mechanism as discussed in Sect. 4.3, this coherence protocol whose update events have unique timestamps is consistent with the virtual time and satisfies the exact causal ordering of updates to the shared variables.

Transaction-level modeling (TLM) with its well-known definition [10] cannot be used for distributed simulation in a PDES language like PSML. We believe that future hardware languages, new languages for standardization or parallel language extensions

to legacy HW languages, must adopt this strategy and redefine the semantics of the shared state such as TLM interfaces. To facilitate a basic TLM coding style in PSML, we propose the concept of *transactional PDES events* (TPEs), in which a master ULP hosts a TLM-like interface and other ULPs can query the master by exchanging TPEs. TPE is a generalization of PDES events that contains an additional data payload, which can be used to exchange any desired data and implement high-level message-based transactions. As seen, PSML tries to construct an object-oriented logical process MoC that has some language similarities with existing SLDLs and HDLs, but it is only meant to facilitate the learning process of a new user to the PDES world. PSML supports the notion of sensitivity lists that must explicitly be registered to ULPs. Because distributed simulation means communication between processors that may be on different nodes in a network, PSML defines its own composite data type system. All non-primitive PSML data types must inherit from *serializable_object* and implement a number of methods such as *serialize, deserialize, clone*, *copy* and *delete*. PSML implements a broad range of HW data types on top of this system such as 32, 64 and 512 bit integers.

- *Language support for efficient PDES execution* Generally, PDES execution is well suited for fine-grained applications; however, very fine granularity may reduce performance significantly in some scenarios. PSML language and its kernel implement a unified interface for giving the capability of model partitioning, processor mapping, and extraction of runtime information for efficient partitioning to the programmer. After PSML elaboration phase, PSML constructs a network of LPs as a WPG by the port/proxy-based software design pattern. A user can implement different static partitioners (fine-grained and coarse-grained) which are available in the theory of general-purpose parallel computing and PDES. PSML records a large number of useful information at runtime for sequential or parallel simulation, such as communication pattern of the LPs and processor cores, the number of rollbacks and deadlocks, memory usage, etc. Then, the user can use this information to update WPG and implement a user-defined partitioner to partition the model. A similar technique is used to develop the processor mapper to map partitions to processors; this new graph contains the distance of cores as well. PSML is based on LPs, and there is no constraint for this pattern. Furthermore, PSML provides an API for dynamic load-balancing techniques to migrate LPs across processors based on runtime statistics information by the modeler. This extension is only supported by optimistic protocols. Because the underlying parallel simulation is based on the message passing between PSML ULPs, which reside on different cores, it is necessary to support efficient memory allocation. A single unified multi-threaded memory allocator should be accessible to the system-level modelers, who make use of that to extend either the composite data type system or perform high-level transactions between processes, and PSML kernel as well. This functionality is provided by *psml_new* and *psml_delete* keywords. The underlying allocator implements two types of memory pools: one used for intra-cluster and another for inter-cluster communication. It is aware of the sender and receiver ULPs through the API. Therefore, each cluster has a local pool and a shared remote-access pool. Because multiple PSML ULPs usually reside on a single cluster, the first memory pool provides a lockless, fast access to the pool. Spin locks are used for the second type of the pool

in order to protect concurrent access to it, where a sender ULP allocates a chunk of the memory and a receiver ULP frees it in another processor. Both pools are implemented based on stacks to improve cache reuse. Each pool itself is a hierarchical pool of memory pools where each sub-pool maintains a separate free list for each size memory block used for different hardware data types.
- *Language support for LP state specification and management* Unlike legacy PSLs and like low-level PDES libraries, PSML gives full control over state management to the programmer. Since an efficient optimistic execution of a model requires proper model state management, PSML takes this important issue into consideration. As there are many state-saving schemas, PSML must introduce a tradeoff between ease of programming and performance. Giving the explicit control of state saving to the user has the potential for maximum simulation performance. An explicit state can be defined for a stateful ULP by inheriting from *psml_state*. The user must override *copy_state*, *clone_state* and *delete_state* methods of this class. PSML kernel makes use of these methods to support different state-saving schemas in optimistic simulations.

4 PSML parallel execution semantics and its implementation

In this section, we first formally define the simulation semantics of PSML and then discuss its real implementation.

4.1 Parallel simulation execution semantics

This section defines a parallel software-based abstract machine for PSML execution. This model represents an abstract view for parallel execution of HW languages, which works on LP pattern with asynchronous PDES protocols. This machine is made up of two parts: (1) the static part, determining the components of PSML language and its LP states, and (2) the dynamic part, which formalizes behavior of the abstract machine in two sub-layers of the PSML execution semantics (PSML abstract kernel) and LP worldview (PDES abstract machine). The behavior of the abstract machine relies on some operations that operate over the static components.

Definition A.1 (*PSML user logical process*)
$ULP \triangleq \langle S, T, M, XSet, YSet, ESet, fp, fiber, KLP, D \rangle$, where

- S is the set of state variables.
- $T \in \{Stateful, Stateless\}$ stands for *ULP* type.
- $M \in \{WithNoArgWait, WithTimedWait, WithEventWait\}$ stands for mode that shows to which type of *psml_wait* the *ULP* has suspended itself.
- $XSet$ and $YSet$ are a set of input and output ports. After elaboration phase, an instance of an appropriate pair of connector/interface is associated with each port. *ULP* can be set to be sensitive to *XSet*.
- $ESet$ is a set of PSML events registered by *psml_wait*(*eventList*) which express the dynamic sensitivity of *ULP*.
- fp is the address of a method to the *ULP* behavior extracted by the PSML compiler.

- KLP is a PDES kernel logical process associated with a ULP.
- $fiber$ is a continuation storage stored in LIFO order that is used to suspend and restore ULP. This is typically implemented by fiber of execution.
- D stands for delay registered for ULP.

Based on Definition A.1, we can also define a hierarchical $Model_{psml}$ as a set of components and proxies where each component may contain a set of sub-components. We omit to give it due to space limitation.

Definition A.2 (*Kernel logical process*)
$KLP \triangleq \langle StateSet,\ EventSet,\ LVT,\ \varphi^S,\ \varphi^\beta,\ INIT \rangle$, where

- $StateSet$ is a set of state variables.
- $EventSet$ is a set of event types associated with the ULP.
- LVT is the local virtual time.
- φ^S stands for state transition function.
- φ^β is event-scheduling function.
- $INIT$ stands for initial configuration of $StateSet$, port/interface binding to the ULP, and initialization of the interfaces.

The above definition is consistent with the majority of existing asynchronous PDES platforms where LPs don't share state variables and assume state space partitioning. Each LP processes one or more event(s) by φ^S and generates new event(s) by φ^β. PSML abstract kernel performs the execution of a system-level model under two phases: PDES-aware elaboration and parallel simulation. Elaboration phase checks PSML model syntactics and constructs the design hierarchy and data structures that are needed for parallel execution atop PDES abstract machine. In elaboration phase, PSML logical process network (LPN) is created by traversing the model hierarchy through ports and proxies with the help of connectors. Then, each ULP is mapped to a KULP, and LPN partitioning into clusters is performed.

Definition A.3 PSML logical process network (LPN) and extended LP network ($\overline{LPN}$)

A.3.1 $LPN \triangleq \beta\big(Model_{psml}\big) = \{ULP_1,\ \ldots,\ ULP_n\} = G(V,\ E)$, where β is the first stage of elaboration phase, $n = |LPN|$, G is the resultant graph of ULPs in which V is a set of PSML processes, and E is set of a pair of target/destination ports where each port stores its process reference and its type.

A.3.2 $\overline{LPN} \triangleq \gamma(LPN) = \big\{\overline{KLP_1},\ \ldots,\ \overline{KLP_n}\big\}$, $\overline{KLP_i} \triangleq \langle KLP_i,\ ULP_i \rangle$, where γ is the second stage of elaboration phase to generate LP network.

A.3.3 $S_{psml} = S_{LPN} \triangleq \big\{s_{\mathrm{ULP}_1},\ \ldots,\ s_{\mathrm{ULP}_n}\big\}$, $S_{\overline{LPN}} \triangleq \Big\{s_{\overline{\mathrm{KLP}_1}},\ \ldots,\ s_{\overline{KLP_n}}\Big\}$, where S is state space.

Examples of s_{ULP} are content of physical interfaces and state of a stateful ULP. A s_{KLP} stores the variables that are used by PSML kernel to manage the execution, for instance, the type of a ULP or to which *psml_wait* the ULP was registered to suspend itself. After preparing the necessary underlying environment for parallel simulation through mapping the model hierarchy into the PDES elements, PSML kernel performs the simulation execution phase. Execution phase consists of three distributed stages

of initialization, parallel simulation and cleanup. We formalize the semantics of the PSML execution phase by using the structural operational semantics (SOS) of a PSML model. It is defined on the parallel abstract machine by an extended version of the labeled-transition system (LTS).

Definition A.4 (*Timed LTS*)
$TLTS \triangleq \langle S, S_0, L, \rightarrow \rangle$, where S is a set of state spaces, $S_0 \in S$ is initial state, L is a set of actions related to event types (*EventSet*), and $\rightarrow \subseteq S \times (L \times Time) \times S$ is the transition relation with delay. An action $a \in L$ is defined as the triplet $a \triangleq \left\langle et, \varphi_{et}^{s}, \varphi_{et}^{\beta} \right\rangle$ where et stands for event type. A typical timed state transition of the system is expressed by $S \xrightarrow{a, delay} S' \equiv \langle S, (a, delay), S' \rangle \in \rightarrow$.

Now, SOS rules are derived for execution of the PSML abstract kernel.

Definition A.5 (*The SOS of a PSML Model*)
$TLTS(PSML) = TLTS(\overline{LPN}) \triangleq \langle System, System_0, L, \rightarrow \rangle$, where

- $System \triangleq \langle S, LVT, \Delta, \nabla, Mode \rangle$
  1. $S = [s_1 \ldots s_n]^T : s_i$ is the state space of $\overline{KLP}_i$.
  2. $LVT = [lvt_1 \ldots lvt_n]^T : lvt_i$ is the local virtual time of $\overline{KLP}_i$. The simulated time of the entire system can be computed as $Time = Min_{i \in n}(lvt_i)$ at any moment in time.
  3. $\Delta = [\Delta_1 \ldots \Delta_n]^T : \Delta_i$ is the future event list (FEL) of $\overline{KLP}_i$. $\Delta_i.min$ means the event with minimum timestamp in Δ_i. $\Delta_i.min.Type$ stands for event type and $\Delta_i.min.id$ is the receiver LP.
  4. $\nabla = [\nabla_1 \ldots \nabla_n]^T : \nabla_i$ is the unsent scheduled event list of $\overline{KLP}_i$ that is sorted in FIFO order.
  5. $Mode = [m_1 \ldots m_n]^T : m_i \in \{Initializing, Processing, Scheduling\}$ denotes the next expected mode of an $\overline{KLP}_i$ after current mode.
- $System_0 = [INIT_1 \ldots INIT_n]^T$ is the initial state of all $\overline{KLP}$s.
- $L = \{L_1, \ldots, L_n\}$: L_n is a set of actions that is defined for each event type in *EventSet*.
- The transition relationship $\rightarrow$ is defined by the following semantics:
  1. Event processing at $\overline{KLP}_i$

$$\frac{(mode_i = Processing) \wedge (\Delta_i.min.Type \in EventSet) \wedge (\Delta_i.min.id = i)}{\langle s_i, lvt_i, \Delta_i, \nabla_i, m_i \rangle \xrightarrow{(et, \varphi_{et}^{s})} \langle s_i', lvt_i', \Delta_i', \nabla_i', m_i' \rangle}$$

where $(s_i' = \varphi^{s}_{\Delta_i.min.Type}(s_i), lvt_i' = \Delta_i.min.timestamp, \Delta_i' = \Delta_i - \Delta_i.min, \nabla_i' = G(ULP_i)|_{t=lvt_i}, m_i' = Scheduling)$.
G is a function that gathers the events generated during the execution of ULP_i by $\varphi^{s}_{\Delta_i.min.Type}$. The third term inside the premise is because multiple LPs may reside on a single cluster and share the same FEL.

2. Event scheduling at $\overline{KLP_i}$

$$\frac{(mode_i = Scheduling) \wedge (\nabla_i \neq \emptyset)}{\left\langle \begin{bmatrix} s_i \\ S_{\nabla_i} \end{bmatrix}, \begin{bmatrix} lvt_i \\ LVT_{\nabla_i} \end{bmatrix}, \begin{bmatrix} \Delta_i \\ \Delta_{\nabla_i} \end{bmatrix}, \nabla_i, m_i \right\rangle \overset{\left(et, \varphi^{\beta}_{et}, delay\right)}{\longrightarrow} \left\langle \begin{bmatrix} s_i \\ S_{\nabla_i} \end{bmatrix}, \begin{bmatrix} lvt_i \\ LVT_{\nabla_i} \end{bmatrix}, \begin{bmatrix} \Delta_i \\ \Delta'_{\nabla_i} \end{bmatrix}, \nabla'_i, m'_i \right\rangle}$$

where $(\forall x \in [0, |\nabla_i| - 1] \subseteq \mathbb{N}^0,\ \Delta'_{\nabla_i[x].id} = \Delta_{\nabla_i[x].id} \cup \{\varphi^{\beta}_{\nabla_i[x].Type}, lvt_i \hat{+} delay >\};\ \nabla'_i = \emptyset, m'_i = Processing)$.
Here, S_{∇} stands for the set of state spaces that are specified by the receipt LPs in ∇, namely, $S_{\nabla} = [\nabla[0].id.s \ldots \nabla[m-1].id.s]^T$ and $m = |\nabla|$. LVT_{∇} and Δ_{∇} are defined similarly. To cope with simultaneous events lvt_i is defined as a composite time and summed with the *delay* by the special operator $\hat{+}$. The notation $\hat{+}$ will be derived in Sect. 5.3.

At last, we turn our attention to describe three remaining semantics for $System_0, \varphi^s_{et}$ and φ^{β}_{et}. Initialization stage denoted by $INIT_i$ for each $\overline{KLP}$ is defined in Definition A.6.

Definition A.6 The Rule for $INIT_i$

$$\frac{(mode_i = Initializing)}{\langle s_i, lvt_i, \Delta_i, \nabla_i, m_i \rangle \rightarrow \langle s'_i, lvt'_i, \Delta_i, \nabla'_i, m'_i \rangle}$$

where $s'_i = SwitchToFiber(ULP_i.fiber)|_{s=s_i}$, $lvt'_i = 0$, $\nabla'_i = G(ULP_i)|_{t=0}$, $m'_i = Scheduling$.

PSML kernel can switch between ULPs; however, the local state of a ULP must be stored to switch between LPS. Fibers provide this feature. They are actually lightweight, userspace threads of control that enjoy from cooperative multitasking. There are two types of fibers: one for clusters and another for ULPs. For instance, we can cooperatively come back to a cluster of the PSML kernel by using these two fibers after transferring from a cluster to a ULP. At this stage, all logical processes are executed once on all of the clusters by PDES abstract machine, and then, parallel simulation begins. Simulation control is transferred from PSML kernel to the model process by switching from the cluster fiber to that LP fiber.

After initialization, parallel simulation begins. PSML parallel scheduling semantics is presented in terms of the two functions φ^S and φ^{β}. An LP receives and processes three types of events in *EventSet*. Therefore, we define φ^S as $\varphi^S = \{\varphi^S_1, \varphi^S_2, \varphi^S_3\}$ shown in Fig. 9. The first event type is concerned with the data update of a port connected to a PI instance that is recognized by the event *PI_EVENT*. Index of the input port to the desired LPs is extracted from the field *port* of the input event which is used to find the interface bound to the port. If the optimization of zero-copy intra-cluster communication is enabled, the method *commit* is called for sender and receipt processes that are on the same cluster.

The sender process does not directly update its internal contents that in turn are committed by the receipt LP upon receiving its related event through this method. Otherwise, the data contents inside the event are copied into the interface through

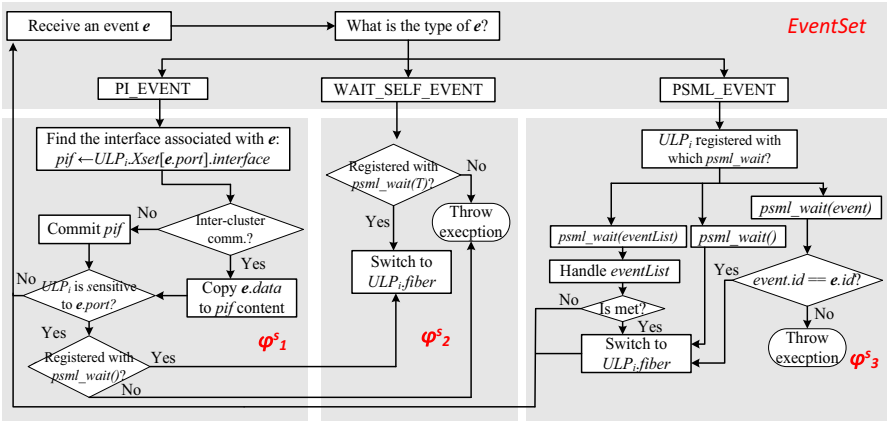

Fig. 9 Algorithm of the state transition function φ^S with respect to event types in $EventSet$

calling the method *copy*. All memory operations are performed through the memory allocator discussed in Sect. 3.2. After preparing the content of the interface, if the ULP is sensitive to the input port, the execution control is given to that system ULP based on whether the ULP is stateful or stateless. This is a decentralized execution semantics as opposed to the central evaluate/update paradigm in SLDLs. If the ULP has already been registered to a *psml_wait* without any input argument, the kernel turns the execution over the ULP to where it had previously suspended itself by switching to its fiber and the LP execution is resumed. PSML kernel throws exceptions in encountering a situation that is not defined by PSML execution semantics and its specification. The second type of events is *WAIT_SELF_EVENT*, which is created merely by invoking the timed function *psml_wait*(T). The third type of events is PSML system-level event, namely, PDES event. For example, Fig. 9 shows operations that manage the *psml_wait* on a single event issued through *psml_wait*(*event*). To take this event to happen, in addition to the kind of the event, the identifier (ID) of received event must be equal to the event ID that has previously been registered with the ULP. If these conditions are met, the control of execution is transferred to it. For more complex combinations of PSML events that are made up of multiple events, PSML implements some similar functions to handle them. For instance, providing that a *wait* has been issued on multiple concurrent events, a vector is created and registered with the ULP. If an event is received and its ID is in the vector, a local counter associated with the ULP and initially set to zero is incremented. Execution of a ULP (G) may generate a set of events that are stored in ∇_i, for example, writing to an interface or invoking *psml_wait* and *notify*. φ^β is defined for each type in *EventSet*.

For example, when a write is issued at ULP_i, $G(ULP_i)$ records generated events in ∇_i based on receiver ULPs by input ports bound to the interface through its connector. Definition A.7 shows the rules to prepare a PDES event for $\nabla_i[x]$, where x is defined in Definition A.5. Because both issuer and receipt ULPs may reside in a single cluster, the algorithm takes advantage of the zero-copy intra-cluster communication.

Definition A.7 Event-scheduling function
φ^{β} for PI Events$\varphi^{\beta}_{\nabla_i[x]} \triangleq \langle PI_EVENT,\ Port,\ Data \rangle$, where

- NSI_EVENT indicates the type of an event generated by writing to a physical interface.
- $Port$ stands for identifier of the input port located at $\nabla_i[x].receiver$, $Port = \nabla_i[x].receiver.port$.
- $Data$ is the content associated with $\nabla_i[x].interface$. For intra-cluster communication $(KLP_i.cluster == \nabla_i[x].receiver.cluster) \rightarrow Data = \emptyset$; for inter-cluster communication $(KLP_i.cluster \neq \nabla_i[x].receiver.cluster) \rightarrow Data = \nabla_i[x].interface.clone()$.

4.2 Implementation of the PSML parallel abstract machine

PSML kernel has been implemented atop a set of objected-pointed PDES-compliant APIs that export a comprehensive, flexible and cross-platform pattern of logical processes with event scheduling world view. It implements the PSML abstract kernel. PSML compiler emits C++ description of a system-level model written in PSML language that conforms with PDES methodology and PSML simulation kernel APIs. PSML kernel is responsible for parallel execution of this native C++-based model. The implementation of the PSML kernel is done on top of a general-purpose PDES wrapper which works based on the LP pattern. This wrapper separates low-level PDES details (e.g., synchronization algorithms, multi-core optimization techniques, and the type of communication, including shared memory and message passing over a distributed network) from PSML kernel and allows it to get a notion of PSML model of parallel computation. It defines the semantics of the PDES abstract machine. Figure 10 shows the PSML design class diagram. SLDL- and PDES-specific classes that comprise the PSML syntactics are used to prepare the simulator internals. We have developed a set of utilities for modelers by extending the PSML kernel APIs. The PS2 wrapper enables the PSML simulation kernel to seamlessly work with different PDES simulation kernels. We have identified minimum requirements that allow PSML to work on several PDES platforms. The top of Fig. 10 depicts this set. These features are encapsulated by PS2 wrapper.

A simulation model compliant with this wrapper consists of three main classes of logical process, state and event. Each logical process has a single state and interacts with other LPs by consuming and generating events (*GetNextEvent,* and *SendEvent*, which are used by φ^{S} and φ^{β}) which have timestamps. This state must specifically be used to interact with optimistic algorithms (*AllocateState*, *GetState* and *CopyState*). An LP processes events (*ExecuteProcess* which is used by φ^{S}) and often modifies its state based on that processed event. A simple time integer is not suitable for parallel simulation of system-level models and SLDLs like PSML. For this purpose, the time class provides a three-tuple representation of time as stated in Sect. 4.3 (*PdesTime* class). This is used to construct a complex time model to implement PSML simulation kernel semantics and to deal with the issue of simultaneous events. To effectively partition the model, each LP should store its connections to the neighboring LPs in a map, where it should not lose the SLDL structural information such as to which

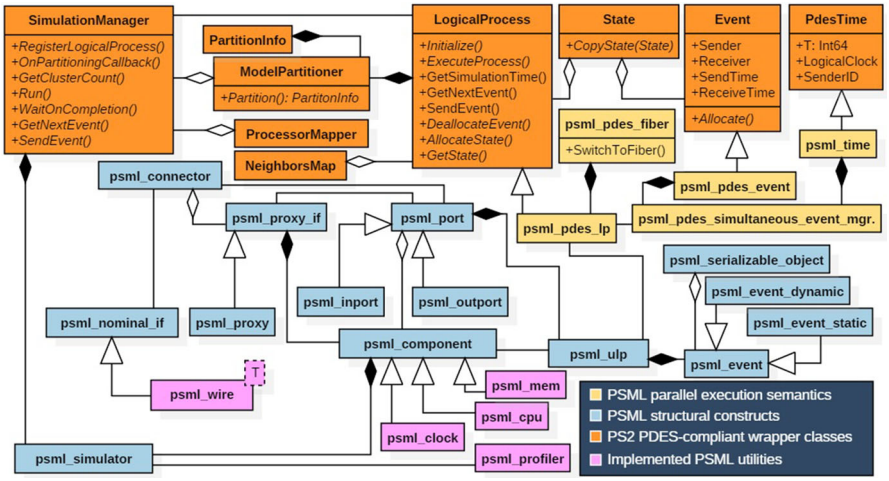

Fig. 10 UML class diagram of the PSML simulation kernel

process and component this LP belongs, and proxies and ports (*NeighborsMap* and *PartitionInfo* classes). The wrapper uses the *ModelPartitioner* and *ProcessorMapper* classes to perform model partitioning and processor mapping prior to the mapping of system-level LPs into the processing elements of the PDES kernel. It also should facilitate PDES-aware elaboration phase of the SLDL kernel. A callback has to be registered with the wrapper, when the partitioning is completed by the PDES kernel so that it can be called to help build the internal data structures of the SLDL kernel (*OnPartitioningCallback*). For example, it is essentially used by the PSML kernel for deciding whether to call *create_instance* method in order to instantiate new instances of nominal interfaces for input ports that come from other processors or not. Furthermore, we employ it to prepare fibers of execution, which realize cooperative scheduling of ULPs, for each partition. Parvicursor Simulation System (PS2) as shown in Fig. 11, which is a new PDES framework, is implemented using multi-core services provided by Parvicursor infrastructure and fully complies with the wrapper.

SimulationManager class was extended to implement an asynchronous conservative simulation kernel that supports the deadlock detection and recovery (DDR) algorithm on multi-core and many-core systems. Each cluster is assigned to one worker thread, and a controller thread is used to recover from deadlocks. Algorithm 1 shows the main loop of each worker thread. PS2 lets the deadlocks occur and benefits from a mechanism to detect them and to recover from them. Deadlocks are broken simply by a global lock-free concurrent counter that is indicating the number of running processes in shared memory architectures. This counter is decremented by one when a process is going to block on an empty input channel and is incremented by one when a process unblocks. At the beginning of simulation, this counter is set to the number of clusters that are hosted by worker threads. When a cluster sees that the counter is zero, a global deadlock is detected and the controller thread is awakened up. In this situation, the controller finds those clusters that contain events with the global minimum timestamp, and deadlock is recovered by allowing those clusters to process such events.

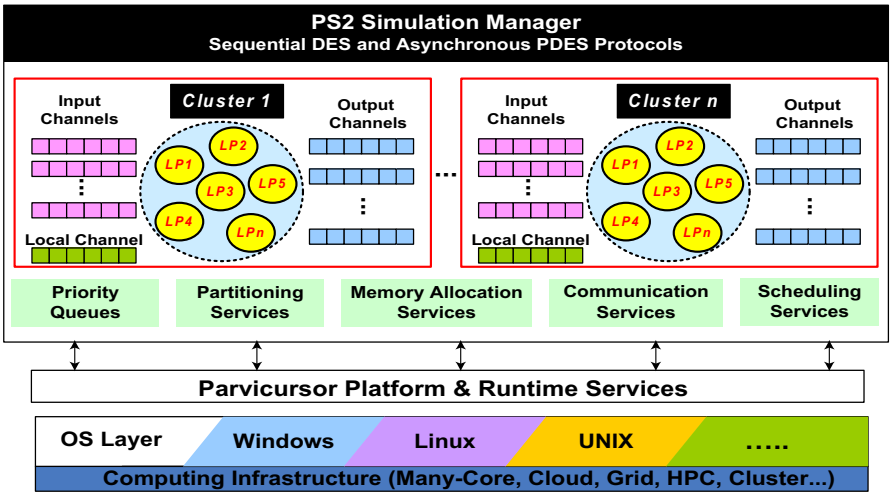

Fig. 11 The general architecture of PS2 framework

Algorithm 1. Main loop of the worker threads in PS2.

Definitions:

1 *Event: e,nextEvent*
2 *PdesTime: now*
3 *T* is the amount of completion time. Each thread owns a local priority queue, a number of input FIFO channels, and a set of mutexes and condition variables for these channels.

The Algorithm:

4 Initialize all LPs belonging to current cluster.← Note: *Initialize*() from wrapper
5 Barrier
6 **while** *now* < *T* **do**
7 *nextEvent* ← Peek an event from local priority queue.
8 **for all** *ch* ∈ input channels **do**
9 Acquire *ch.lock*.
10 If *ch* is empty, decrement global atomic counter; if counter is zero then wake up the controller thread and put current thread into sleep by *ch.condVar*.
11 *e* ← Peek an event from *ch*.
12 Release *ch.lock*.
/ If e is related to a detected deadlock, we locally handle it for this cluster */*
13 **if** *e.type=DEADLOCK_DETECTED* **do**
14 *nextEvent* ← Fetch the event with minimum timestamp from *ch* reported by controller.
15 Perform the actions in lines 17 to 19, and then go to line 6.
16 **if** *e.receiveTime* < *nextEvent.receiveTime* **do** *nextEvent* ← *e*
17 *now* ← *nextEvent.receiveTime*
18 *receiver* ← *nextEvent.receiver, receiver.lvt* ← *now*
/ Process the next event */*
19 *receiver.ExecuteProcess*() ← Note: *ExecuteProcess*() from wrapper
20 Barrier
21 Finalize all LPs belonging to current cluster.← Note: *Finalize*() from wrapper

No successful example has reported to speed up the simulation of digital systems based on DDR protocol since the last 20 years. Because the LP graph resulting from a digital system model is a directed graph, we can detect its cycles and significantly

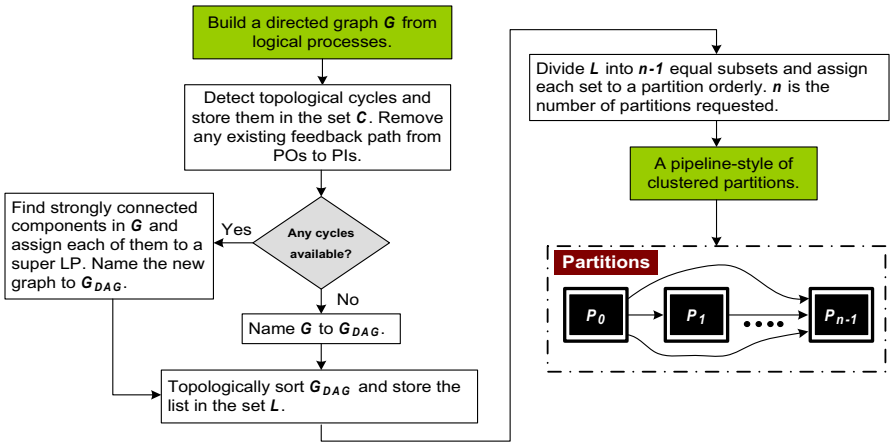

Fig. 12 The PTS partitioning algorithm for DDR synchronization protocol in PS2

decrease the effect of topological deadlocks using a convenient partitioning of the processes into clusters. Most partitioning algorithms have been proposed for optimistic PDES protocols that cannot be used in DDR-based simulations due to the possible creation of artificial topological deadlocks. Pipeline-Style Topological Sort (PTS) algorithm is proposed to solve this problem. The main idea behind PTS algorithm is to treat strongly connected components (SCCs) of an LPN as super LPs and to partition the new graph by applying topological sorting of the reduced directed acyclic graph (DAG). Figure 12 describes the algorithm in detail.

4.3 Simultaneous events and reproducibility problem

Simultaneous events, two or more events that are scheduled exactly to occur at the same simulation time, cause many difficulties in sequential and parallel simulation. The order of such events can dramatically affect the outcome of computations and lead to logical validation errors in parallel simulation. We give a general solution for PSML kernel. This problem is solved by a tie-breaking technique to extend the timestamps to have an additional tag for sequential simulation. A global counter is used and incremented by one when inserting an event into the simulator priority queue. The value of this counter is copied into the tag and sorting is done through this compound priority, which satisfies total insertion order. In parallel simulation where LPs execute on different processors, we must consider a much more sophisticated algorithm. Now, a distributed tie-breaking strategy is given that can be seamlessly used with asynchronous distributed PDES protocols, conservative and optimistic. In PSML kernel, the simulation time is defined as the triplet $t = \langle T, LC, ID \rangle$. T is the same timestamp recognized by the system-level modeler as simulation time, LC is Lamport logical clock [48], and ID is the identifier of an LP scheduling an event. Based on this definition for time comparison of two events, we have:

$$t_1 < t_2 \Leftrightarrow T_1 < T_2 \bigvee \left(T_1 = T_2 \bigwedge LC_1 < LC_2\right) \bigvee \left(T_1 = T_2 \bigwedge LC_1 = LC_2 \bigwedge ID_1 < ID_2\right)$$

When events are sorted reliant on this rule, sequential and parallel simulation results are exactly alike and deterministic. LC is automatically computed in each LP when scheduling new events through PS2 wrapper. The two flags of LC and ID are hidden from the user but visible to the PSML kernel and PS2 wrapper. Each LP in PSML kernel retrieves the *t*ransformed time through calling the functions of the simultaneous event manager when new event(s) are sent from its surrounding LPs. *LC* is calculated based on Lamport algorithm in which each LP before sending an event increments a counter associated with the LP. When an LP sends an event, this value is attached to the message. Upon receiving an event, the receipt LP updates its counter to the maximum of its current counter and the received LC and then increments it by one. Each LP has a unique identifier number. This technique is called *total event ordering* or *unique timestamps* which guarantees reproducibility in both parallel and sequential PSML executions. However, if we are willing to see how the PSML simulation results are compliant with existing sequential SLDL semantics, when we are using PSML as a parallel intermediate language, a new algorithm should be developed, because this technique is not necessarily related to the actual chronological order of events in the model. Logical clocks only establish the *happened-before* relations, but LP IDs are assigned by a simple counter when LPs are instantiated. Therefore, the new algorithm should generate identifiers such that SLDL semantics are captured. This algorithm is automatically performed in two steps to calculate IDs. (1) PSML compiler records the order of the triggering points of the events, including writing to wire interfaces, event notifications and *psml_wait* call expressions, in system-level PSML processes by using static code analysis. Then, all expressions and statements with the exception of these points are removed from the model code, and the final code is emitted by the compiler. (2) The generated codes are compiled and linked with the PSML kernel and necessary runtime libraries. Then, sequential simulation is done using a tie-breaking method for simultaneous events with zero lookahead for all triggering points. During simulation, null messages, instead of actual events originated from triggering points, are generated and penetrate into the model execution. Using code instrumentation of the execution, we assign a priority value as the identifier to each LP that is currently executing. Processes that run earlier have lower IDs. Execution completes until all LPs receive their own IDs.

5 Experimental studies

We study the results of a set of experiments to evaluate the performance of PSML framework. Tests were carried out on a 12-core machine operating at 2.67 GHz with 16 GB main memory and 12 MB L2 cache. Linux CentOS with kernel 2.6.32 was installed on this machine. Tests are reported with respect to Accellera sequential reference and parallel SystemC simulators. Speedup is defined as the ratio of the wall times of sequential and parallel execution. Most of the models are automatically partitioned based on PTS. All the tests are an average of 10 runs. High-level architecture of benchmarks is shown in Figs. 13, 14, and 15.

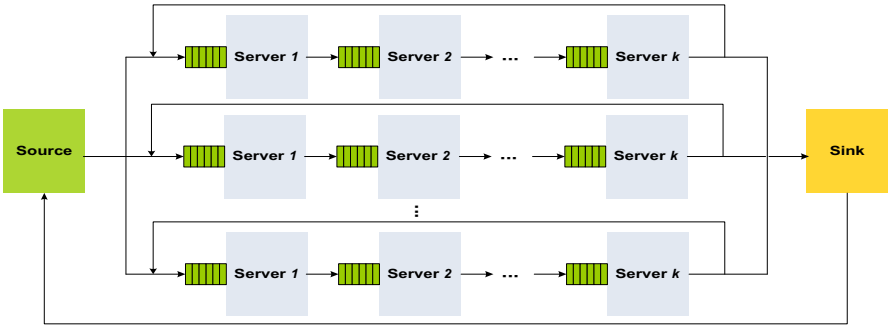

Fig. 13 A pipelined clustered tandem queue

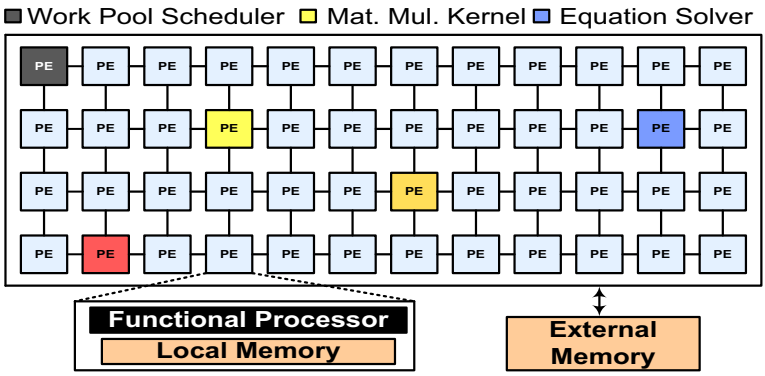

Fig. 14 A many-core model

5.1 System-level experiments

In the first evaluation, we developed three system-level models relied on the PSML language: two pipeline and clustered tandem queues shown in Fig. 13, and a many-core bus functional model (BFM) and TLM illustrated in Fig. 14. All nodes are synchronized with each other through distributed PSML events. In the pipelined experiment, there are 50 thousand servers that are executed along with the source and sink nodes fed by 50,000 different generated patterns. In the clustered pipelined scenario, there are 48 rows each of which have 1024 servers. In this network, 1024 patterns are generated once and circulated in the closed paths shown. Simulation results are illustrated in Fig. 16 and Table 2. This diagram represents the speedups depending on the number of logical OS kernel threads in the multi-level PSML thread scheduling.

The many-core BFM architecture is organized as a 2D rectangular mesh of homogeneous cores shown. Each Processing Element (PE) consists mainly of a simple processor and a local memory. The processor is modeled as a functional behavior of a kernel solver such as matrix multiplication, Sobel and FIR filter, Adaptive Differential Pulse-Code (ADPC), JPEG Encoder, and MD5. They are purely written in C# methods in PSML as snippets of software embedded in the system-level model.

Table 2 Execution times for different benchmarks

Model	Sequential SystemC (s)	Sequential PSML (s)	PSML parallel execution time (s) based on kernel threads (KT)				
			8 KT	16 KT	24 KT	48 KT	138 KT
pipeline	745.78	820.23	212.45	104.16	76.34	51.13	40.81
clustered	26,304.12	29,971.92	3704.71	1814.59	1594.17	1453.95	1362.36
AES	826.15	853.78	155.16	92.85	77.37	50.49	36.17
QSORT	423.81	492.25	59.13	37.54	26.59	19..35	12.95
b18	5231.4	5848.172	951.19	601.41	528.93	337.38	208.12
s35932	220.37	241.68	45.83	27.2	24.74	16.34	9.9
b22	3713.56	3992.65	571.91	322.14	273.64	229.43	207.18
MatMul(48)	102.5	90.47	12.9	12.01	7.5	9.1	12.6
SOBEL	150.57	135.08	39.54	20.03	10.43	5.54	6.43
APDC	100.46	90.75	12.4	8.81	6.78	7.49	6.48
FIR	86.26	79.69	10.39	5.94	4.1	4.67	6.13
JPEG	674.32	664.93	192.66	89.9	49.81	30.23	45.16
MD5	243.19	236.51	26.14	19.30	15.4	16.65	18.29

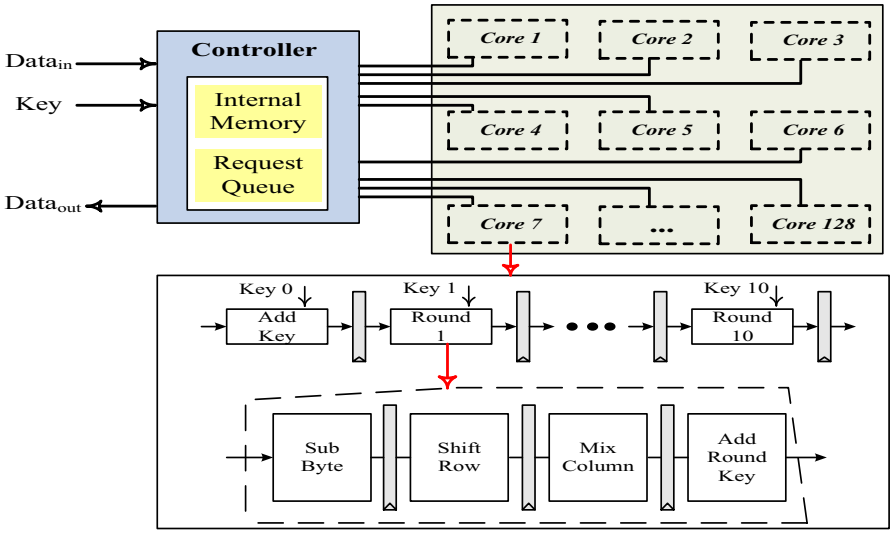

Fig. 15 A simplified multi-core AES accelerator

Communication between PEs is performed through bus channels as PSML wires, which are cycle-accurate and pin-accurate. The estimated time of computation of each functional processor is annotated into the code by inserting *psml_wait* statements. A central PE, the work pool scheduler, controls the execution of other PEs. It distributes the works among the computing kernels. Each kernel copies the information of it assigned task from external memory to its local memory before executing the task. Figure 17 illustrates the speedups for different deployments of the BFM models. A

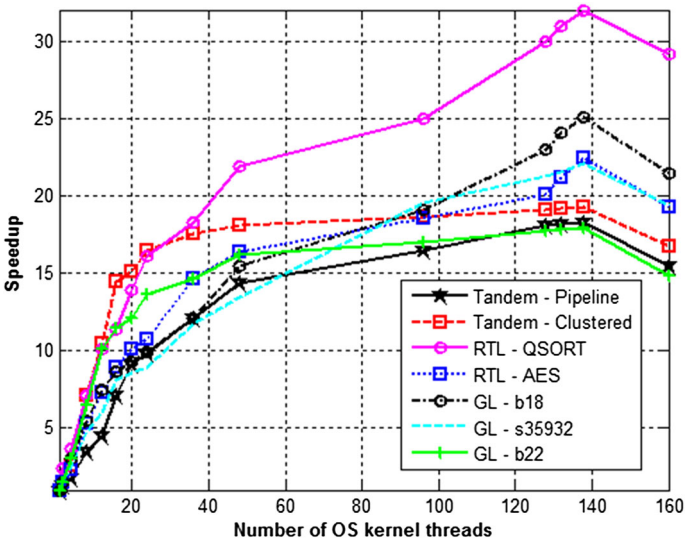

Fig. 16 The speedups of different benchmarks

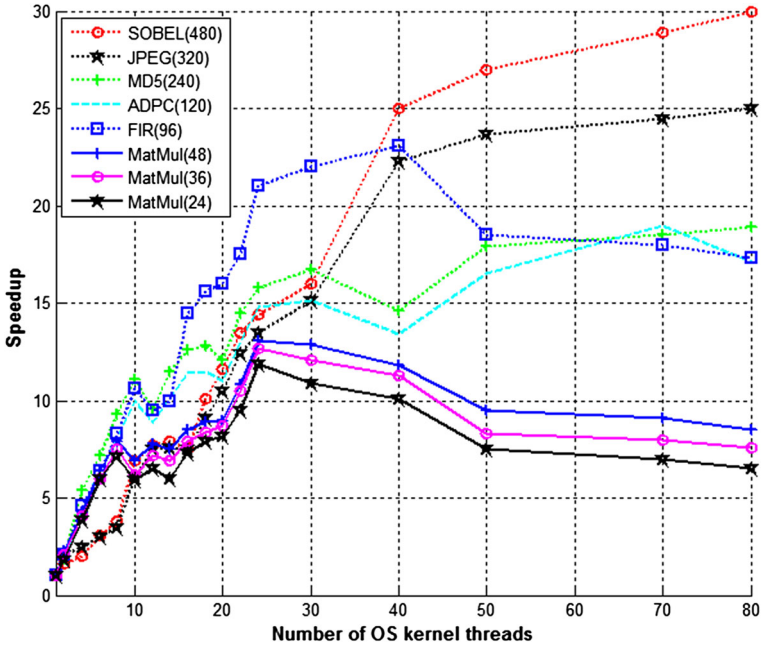

Fig. 17 Speedups of the many-core BFM model

customized model partitioner was developed for this test, where a number of PEs is assigned to a single partition. External memory and the master clock process are also allocated to different partitions.

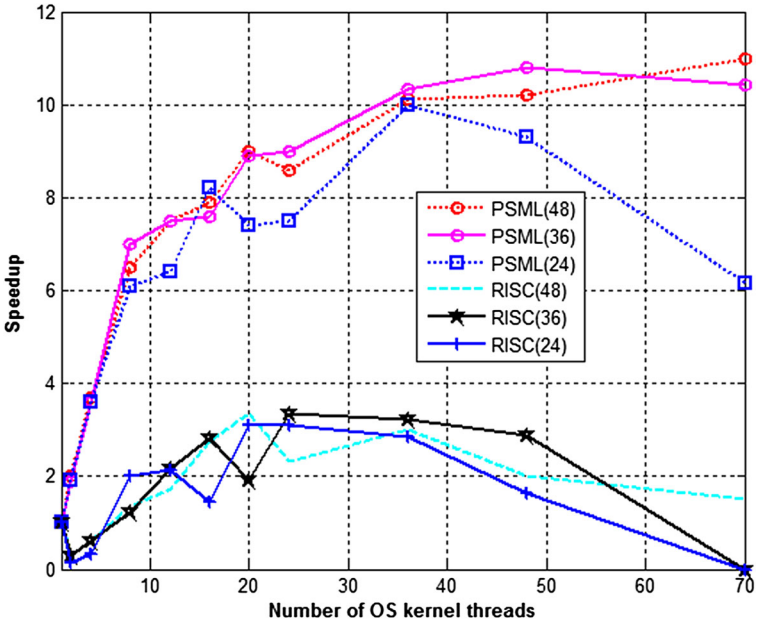

Fig. 18 Speedups of the many-core TLM model

To compare the performance of PSML and an existing parallel SystemC simulator, we chose the work in [26, 33, 36–38, 49] called RISC, which is the only accessible parallel SystemC online. It implements an improved version of shared memory synchronous PDES and protects shared variables such as TLM channels by mutexes. Since RISC does not support a number of SystemC features such as the sensitivity lists, we had to re-implement the many-core BFM model to work based on FIFO TLM channels. In PSML due to asynchronous simulation, each FIFO interface is hosted by a separate ULP and TPEs are used to communicate PEs with the FIFO ULPs. Figure 18 and Table 3 show the speedups, and compile and execution times for both platforms. The underlying RISC synchronization algorithm benefits from a compile-time static analysis phase to detect the events that have no interdependency, and to execute them at runtime for improving the execution speed. For models with more processes, the compile time grows exponentially by many hours.

5.2 Register transfer level experiments

In the second set of tests, we examine two developed RTL benchmarks. Two different single-core models were translated manually from an existing codebase written in Verilog into PSML, including a cryptography algorithm based on AES and a sorting algorithm for integer lists. The AES model had been implemented as a pipeline IP core. We extended it and developed a multi-core AES accelerator in PSML. Figure 15 illustrates the fully pipelined architecture of this accelerator with 128 cores. Figure 16 shows the performance of the circuits.

Table 3 Comparison between PSML performance and an existing parallel SystemC implementation called RISC

Model	Compile time (s)		Sequential SystemC (s)	Sequential PSML (s)	Parallel execution time (s)					
					16 KT		24 KT		48 KT	
	PSML	RISC			PSML	RISC	PSML	RISC	PSML	RISC
TLM (48)	3	175	317	313	40	115	37	95	31	103
TLM (36)	3	111	238	235	32	84	26	71	23	83
TLM (24)	3	71	165	163	20	114	22	51	18	113

5.3 Gate-level experiments

In the last experiment, we evaluate GL tests. Figure 16 depicts the speedups of three ISCAS (International Symposium on Circuits and Systems) benchmarks with different sizes and functionalities. The lookahead of all gates was set to one. In these tests, all gates are modeled as stateless processes and the circuits are fed by a pattern generator component.

5.4 Discussion on PSML simulation speedups

As the measurements promote super-linear speedups are dominant on 12 physical cores. They are mainly due to the cache effect available in different memory hierarchies. As seen, this effect discloses itself at high number of OS kernel threads for most of the models. The main reason can be attributed to the granularity control, and optimizations applied to both PSML and PS2 simulation kernels that in turn greatly result in decreasing overhead and the idle time of physical cores. As stated earlier, PS2 benefits from a decentralized (distributed event list management), peer-to-peer architecture; namely, all components of the simulator are distributed over physical cores, and the use of shared components is minimized. Each communication channel has one lock and one condition variable, and they are also distributed in the entire simulator. Load balancing is obtained by precise partitioning of the model. Therefore, resource contention is avoided as much as possible. PSML simulation kernel implements a multi-level multi-threading scheduler that allows multiple system-level processes are assigned to a single cluster and each cluster is mapped onto a single OS kernel thread. Each physical processor can thus host multiple kernel threads. In DDR execution due to conservative nature, additional kernel threads can give rise to reduce the blocking time of kernel threads on physical cores, because OS thread scheduler switches between different threads and lets other threads (likely unblocked threads) run on that core.

Adding much more kernel threads have a significant negative effect on speed, which indicates load imbalance, increase in context switches of kernel threads and cache pollution due to aggregate thread stacks. The underlying PSML model of computation facilitates extremely fine-grained partitioning at the level of process network at runtime which lets the partitioner robustly provide load balancing of processes over cores. Our measurements verify that increasing the number of logical threads in PSML reduces deadlocks for larger models on each core. Optimized use of the memory management has also other important impact on performance. Decentralized (distributed memory management) architecture of the PSML memory allocator, like PS2, decreases memory contentions for allocating and releasing events dramatically, while intra-cluster allocations are performed without lock synchronization. The memory usage is related to the number of clusters, which is associated with the space required to maintain the data structures of both simulation kernels as well as the stacks of OS kernel threads. Each model itself consumes a constant amount of memory space proportional to the total of PSML processes. The whole models occupies up to 6% out of the total memory partitioned by PTS. This signifies a good load balance is made by this partitioner. Dis-

tributing an uneven number of processes to cores may cause a fast process to generate events before others can execute them instantly; therefore, extra memory allocations finally arrive at the lack of memory and termination of the simulator. Zero-copy communication is also an important factor to exploit the locality of reference and cache reuse. In PSML, processes residing on the same cluster have direct access to interfaces, where no copy of the interface content is performed when sending intra-cluster events. In PS2 kernel, reference to a message is directly sent without any copy of the original message. It is worth noting that for large-scale models like AES, sequential PSML execution time is slightly larger than sequential SystemC. This is because PSML makes use of the sequential PS2 simulation manager that instead emulates the execution of the LP-based model as an event list-based implementation. Since the sequential simulator maintains additional data structures for the LP pattern, it cannot benefit from cache locality on a single core.

6 Conclusion and future directions

In this paper, we proposed a parallel simulation language for fast modeling and simulation of digital electronic systems. This language, along with the collection of introduced tools, provides the ability to simulate models at different levels of abstraction by using a wide range of PDES synchronization protocols in order for the designer to leverage the massive parallelism of today's multi-core and many-core platforms. It was shown that the introduced architecture improves simulation speed significantly with linear, super-linear speedups ranging from 11 $\times$ to 32 $\times$ for large-scale, complex PSML models on a 12-core host when being run by an optimally implemented asynchronous PDES algorithm. We are implementing an optimistic simulation kernel for PSML. Because PSML, as an intermediate language, is directly applicable to existing SLDLs and HDLs, we intend to develop a comprehensive compiler to transform SystemC models into PSML. We plan to implement a distributed PDES kernel atop many-core HPC clusters.

References

1. Sinaei S, Fatemi O (2018) Multi-objective algorithms for the application mapping problem in heterogeneous multiprocessor embedded system design. J Supercomput 1–27. https://doi.org/10.1007/s11227-018-2442-2
2. Fujimoto RM (2000) Parallel and distributed simulation systems. Wiley, New York
3. Fujimoto RM (2016) Research challenges in parallel and distributed simulation. ACM Trans Model Comput Simul (TOMACS) 26(4):22
4. Sang J et al (2018) Experiences with implementing parallel discrete-event simulation on GPU. J Supercomput 1–18. https://doi.org/10.1007/s11227-018-2254-4
5. Jafer S, Liu Q, Wainer G (2013) Synchronization methods in parallel and distributed discrete-event simulation. Simul Model Pract Theory 30:54–73
6. Bagrodia RL (1998) Parallel languages for discrete-event simulation models. IEEE Comput Sci Eng 5(2):27–38
7. Barnes Jr PD et al (2013) Warp speed: executing time warp on 1,966,080 cores. In: Proceedings of the 1st ACM SIGSIM Conference on Principles of Advanced Discrete Simulation. ACM

8. Society IC (2008) IEEE Standard 1076-2008—IEEE Standard for VHDL Language Reference Manual. IEEE
9. Society IC (2005) IEEE Standard 1364-2005—IEEE Standard for Verilog Hardware Description Language. IEEE
10. Society IC (2011) IEEE Standard 1666-2011—IEEE Standard for SystemC Language Reference Manual. IEEE
11. Society IC (2012) ANSI/IEEE Standard 1800-2012—IEEE Standard for System Verilog–Unified Hardware Design, Specification, and Verification Language. IEEE
12. Dömer R, Gerstlauer A, Gajski D (2002) SpecC language reference manual. In: SpecC technology open consortium
13. Dahl O-J, Nygaard K (1966) SIMULA: an ALGOL-based simulation language. Commun ACM 9(9):671–678
14. Xia W, Yao Y, Mu X (2012) An extended event graph-based modelling method for parallel and distributed discrete-event simulation. Math Comput Model Dyn Syst 18(3):287–306
15. Barros FJ (2008) Modeling and simulation of parallel adaptive divide-and-conquer algorithms. J Supercomput 43(3):241–255
16. Zhu L et al (2005) Parallel logic simulation of million-gate VLSI circuits. In: Modeling, analysis, and simulation of computer and telecommunication systems, 2005. 13th IEEE international symposium on. IEEE
17. Meraji S, Zhang W, Tropper C (2010) On the scalability and dynamic load-balancing of optimistic gate level simulation. IEEE Trans Comput Aided Des Integr Circuits Syst 29(9):1368–1380
18. Zhu Y, Wang B, Deng Y (2011) Massively parallel logic simulation with GPUs. ACM Trans Des Autom Electron Syst (TODAES) 16(3):29
19. Meraji S, Tropper C (2012) Optimizing techniques for parallel digital logic simulation. IEEE Trans Parallel Distrib Syst 23(6):1135–1146
20. Bailey ML, Briner JV Jr, Chamberlain RD (1994) Parallel logic simulation of VLSI systems. ACM Comput Surv (CSUR) 26(3):255–294
21. Gonsiorowski E, Lapre JM, Carothers CD (2017) Automatic model generation for gate-level circuit PDES with reverse computation. ACM Trans Model Comput Simul (TOMACS) 27(2):12
22. Martin DE et al (2002) Analysis and simulation of mixed-technology VLSI systems. J Parallel Distrib Comput 62(3):468–493
23. Li L, Huang H, Tropper C (2003) DVS: an object-oriented framework for distributed verilog simulation. In: Parallel and distributed simulation, 2003 (PADS 2003). proceedings. Seventeenth workshop on. IEEE
24. Sato S, Kobayashi R, Kise K (2018) ArchHDL: a novel hardware RTL modeling and high-speed simulation environment. IEICE Trans Inf Syst 101(2):344–353
25. Chen W et al (2014) Out-of-order parallel discrete event simulation for transaction level models. IEEE Trans Comput Aided Des Integr Circuits Syst 33(12):1859–1872
26. Chen W, Han X, Dömer R (2012) Out-of-order parallel simulation for ESL design. In: Proceedings of the Conference on Design, Automation and Test in Europe. EDA Consortium
27. Schumacher C et al (2010) parSC: synchronous parallel systemC simulation on multi-core host architectures. In: Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis. ACM
28. Vinco S et al (2012) SAGA: SystemC acceleration on GPU architectures. In: Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE. IEEE
29. Reder S et al (2015) Adaptive algorithm and tool flow for accelerating SystemC on many-core architectures. Microprocess Microsyst 39(8):1063–1075
30. Schmidt T, Cheng Z, Dömer R (2018) Port call path sensitive conflict analysis for instance-aware parallel SystemC simulation. In: Design, automation and test in Europe. Dresden, Germany
31. Ventroux N, Sassolas T (2016) A new parallel SystemC kernel leveraging manycore architectures. In: Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016. IEEE
32. Weinstock JH et al (2016) Parallel SystemC simulation for ESL design. ACM Trans Embed Comput Syst (TECS) 16(1):27
33. Schmidt T, Liu G, Dömer R (2017) Exploiting thread and data level parallelism for ultimate parallel SystemC simulation. In: Proceedings of the 54th Annual Design Automation Conference 2017. ACM
34. Cheng Z, Schmidt T, Domer R (2018) SystemC coding guideline for faster out-of-order parallel discrete event simulation. In: Proceedings of forum on specification and design languages. Munich, Germany

35. Doemer R (2016) Seven obstacles in the way of standard-compliant parallel SystemC simulation. IEEE Embed Syst Lett 8(4):81–84
36. Schmidt T (2018) A compiler infrastructure for static and hybrid analysis of discrete event system models. Ph.D. Dissertation, University of California, Irvine
37. Liu G (2017) Optimizing many-threads-to-many-cores mapping in parallel electronic system level simulation. Ph.D. Dissertation, University of California, Irvine
38. Dömer R, Liu G, Schmidt T (2017) Parallel simulation. In: Handbook of hardware/software codesign. Springer, Berlin, pp 533–564
39. Becker D, Moy M, Cornet J (2015) Challenges for the parallelization of loosely timed SystemC programs. In: IEEE international symposium on rapid system prototyping
40. Cox DR (2005) Ritsim: distributed systemC simulation. Department of Computer Engineering. Rochester Institute of Technology
41. Chopard B, Combes P, Zory J (2006) A conservative approach to systemC parallelization. In: Computational science—ICCS 2006. Springer, Berlin, pp 653–660
42. Mubarak M et al (2017) Enabling parallel simulation of large-scale HPC network systems. IEEE Trans Parallel Distrib Syst 28(1):87–100
43. Low Y-H et al (1999) Survey of languages and runtime libraries for parallel discrete-event simulation. Simulation 72(3):170–186
44. Perumalla K, Fujimoto R, Ogielski A (1998) TED—a language for modeling telecommunication networks. ACM SIGMETRICS Perform Eval Rev 25(4):4–11
45. Arora R, Bangalore P, Mernik M (2012) Raising the level of abstraction for developing message passing applications. J Supercomput 59(2):1079–1100
46. Thoman P et al (2018) A taxonomy of task-based parallel programming technologies for high-performance computing. J Supercomput 74(4):1422–1434
47. Poshtkohi A, Ghaznavi-Ghoushchi MB, Saghafi K (2017) The Parvicursor infrastructure to facilitate the design of Grid and Cloud computing systems. Computing 99(10):979–1006
48. Lamport L (1978) Time, clocks, and the ordering of events in a distributed system. Commun ACM 21(7):558–565
49. Liu G et al (2017) RISC compiler and simulator, Release V0.4.0: out-of-order parallel simulatable SystemC subset