A SystemC TLM Framework for Distributed Simulation of Complex Systems with Unpredictable Communication

Julien Peeters, Nicolas Ventroux, Tanguy Sassolas, Lionel Lacassagne
CEA, LIST,
Embedded Computing Laboratory
91191 Gif-sur-Yvette CEDEX, FRANCE
Email: julien.peeters@cea.fr

***Abstract*—Increasingly complex systems need parallelized simulation engines. In the context of SystemC simulation, existing proposals require predicting communication in the simulated system. However, this is often unpredictable.**

In order to deal with unpredictable systems, this paper presents a parallelization approach using asynchronous communication without modification of the SystemC simulation engine. Simulated system model is cut up and distributed across separate simulation engines, each part being evaluated in parallel of others. Functional consistency is preserved thanks to the simulated system *write exclusive* memory access policy while temporal consistency is guaranteed using explicit synchronization. Experimental results show up a speed-up up to 13x on 16 processors.

I. Introduction

Hardware complexity is continuously increasing. For instance, System-on-Chip (SoC's) now integrate ever more sophisticated architectures, targeting multimedia (H.264, VC-1...) or wireless communication (UMTS, WiMAX...). Simulation of such systems slows down as their complexity increases [1].

SystemC is a C++ class library that supports mixed software/hardware system design. System modeling can be done at different levels of abstraction and accuracy. As a consequence, SystemC has become widely used as a tool to explore design space of complex systems. For this task, there is no need for bit-level accuracy. Instead, wire-level communication is abstracted away as high-level transactions implemented by a SystemC extension named *Transaction-Level Modeling* (TLM).

A promising approach is to parallelize simulation. Related proposals [1]–[7] preserve simulation temporal causality thanks to conservative synchronization [8], avoiding the management of an expensive checkpoint/rollback strategy required when using the optimistic variant. However, conservative synchronization becomes a bottleneck when communication is unpredictable. Indeed, such synchronization must occur as soon as a communication can be initiated by any module in the simulated system. In the present context, we assume that this might happen at each simulated system clock cycle, called further *a cycle*. In addition, we will talk, in this paper, about *temporal consistency* which allows a temporal error under certain conditions in opposite to strict causality as defined in previous work.

We focus our work on two key points, which lead to two contributions, addressing simulation and design space exploration of complex unpredictable systems:

- We propose (section II) a parallelization approach, drawing its inspiration from proposals of Mello [2] and Galiano [6]. We partition the system model into *clusters* and evaluate each of them in a separate simulation engine. Communication between clusters is made *asynchronous* so as to avoid blocking simulation. In order to deal with unpredictable systems, we introduce a new *synchronization mechanism* divided in two parts. One relies on the *write exclusive* memory access policy of the simulated system and preserves simulation functional consistency. The other generates explicit synchronization to bound the temporal error introduced by asynchronous communication.
- We implement (section III) our approach as a SystemC TLM framework. Thanks to this framework, the parallelization of a simulation does not imply to rewrite or adapt the simulated system model. Parallelization is *transparent* for the system designer and does not require modifying the SystemC simulation engine. This framework guarantees that functional modeling semantics is preserved when simulation is parallelized.

So as to validate our approach, we build a customizable validation environment (section IV). We use our approach to run many parallel simulations against many environment configurations. The results (section V) offer to characterize the simulation speed-up and accuracy. Furthermore, we deduce from results two simulation modes, which give a simulation speed-up up to 13x on 16 processors compared to a standard non-distributed simulation.

Finally, we compare our approach with related work (section VI) and conclude (section VII) on the results and features given by our approach.

II. Background and suggested new approach

Among promising approaches speeding up SystemC simulation, Mello [2] and Galiano [6] propose to cut up the model describing the system to simulate, also called the *simulated system model*, into clusters. A *cluster* is a partition of the whole simulated system model and then is purely virtual and does not represent a concrete structure in the simulated system. Thereafter, each cluster is evaluated in parallel with each other in a separate simulation engine, running on a separate processor.

Parallel Discrete Event Simulation (PDES) theory [8] provides a formal representation of synchronization in parallel simulation. It has two variants: conservative and optimistic. The optimistic variant lets the simulation speculate on its future states. A rollback mechanism returns to a valid state in case of an incorrect speculation. This requires to record *checkpoints* during simulation in order to roll back to the last valid state when necessary. However, simulation state history becomes difficult to manage while the simulated system complexity increases [5]. In return, the conservative variant does not comprise a speculation nor a rollback mechanism. Instead, simulation is only allowed to progress when synchronization is assured that no past time-annotated communication might occur. This way, it guarantees temporal causality at any time during simulation. For this reason, most of related works implement a conservative synchronization mechanism.

The conservative variant requires knowing the minimum duration between two communications occurring in the simulated system [5], [6]. This duration is called *lookahead* and represents the time during which the parallel simulation can be evaluated without synchronizing. The longer the lookahead, the greater the speed-up. However, in some systems, communication rate cannot be specified. Consequently, lookahead must be shortened to the worst case value: one simulated system clock cycle. This causes the simulation to dramatically slow down or even become unusable.

Our approach draws its inspiration from Mello [2] and Galiano [6] proposals. In our case, communication between clusters is implemented using the Message Passing Interface (MPI) [9]. Our implementation transforms any inter-cluster communication into asynchronous calls. This lets the simulation locally progress on the initiator side while waiting for the communication to complete.

In order to deal with unpredictable systems, we propose a hybrid synchronization approach that is divided in two parts. One part guarantees the functional modeling consistency when a cluster accesses data from another cluster, constraining the simulated system to have a write exclusive memory access policy. This means that a writer has exclusive access to memory when it is writing. Hence, it must wait for all the readers to have finished reading before writing. This constraint creates *implicit synchronization* between clusters. Here, writers and readers are SystemC modules in the simulated system model. However, even if functional consistency is guaranteed, nothing prevents clusters from diverging in time as communication

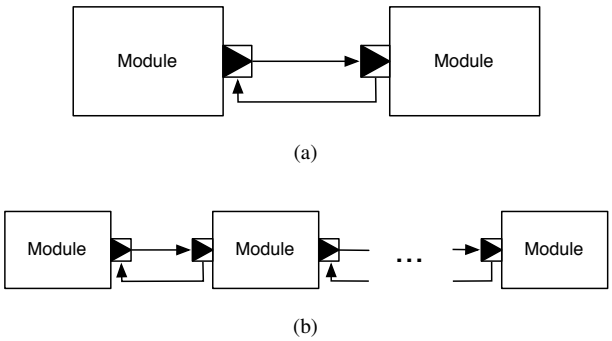

Fig. 1. Example of (a) an initiator connected to a target and (b) chaining of modules. Arrows indicate the direction from an *initiator* to a *target*.

is asynchronous, thereby potentially introducing a *temporal error*. This error varies according to the relative simulation speed deviation between communicating clusters. Then the other part of the synchronization mechanism sends *explicit synchronization* at regular intervals so as to bound this error. The explicit synchronization period is specified by the system designer. The designer can also change the period value so as to tune the simulation accuracy.

We will now present the implementation details of our parallelization approach as a SystemC TLM framework.

III. Framework implementation details

In the TLM specification an initiator initiates a request and sends it to a target. The target processes the request and sends back a response. Such a request-response pair is named a *transaction*. In this context, initiator and target are both SystemC modules (figure 1a). According to the specification, a communication between an initiator and a target may be either *blocking* or *non-blocking*. In the latter case, the response part of a transaction may be omitted. SystemC modules may also be chained. In this case, intermediate modules act like a target for the previous module and like an initiator for the next module in the chain (figure 1b).

When parallelizing a SystemC simulation, the simulated system model is cut up into clusters. Therefore, some communication links between SystemC modules are broken as the result of the cutting. These *broken* links are replaced by *virtual links*, providing the inter-cluster communication substrate. However, the introduction of virtual links does not modify the initial semantics of the simulated system model. Figures 2a and 2b illustrate this transformation.

A virtual link is composed of two end-points named *wrappers*. A wrapper acts both like an interface to the distributed communication layer (i.e. the MPI middleware) and to one of the initiator or target involved in the link (figure 2c). As a consequence, a SystemC module connected to a virtual link ignores whether the link is virtual or not. Then, there is no requirement to modify or adapt initiators and targets for the parallel simulation to work.

Explicit synchronization, introduced in section II, is the central mechanism that prevents cluster's simulations from

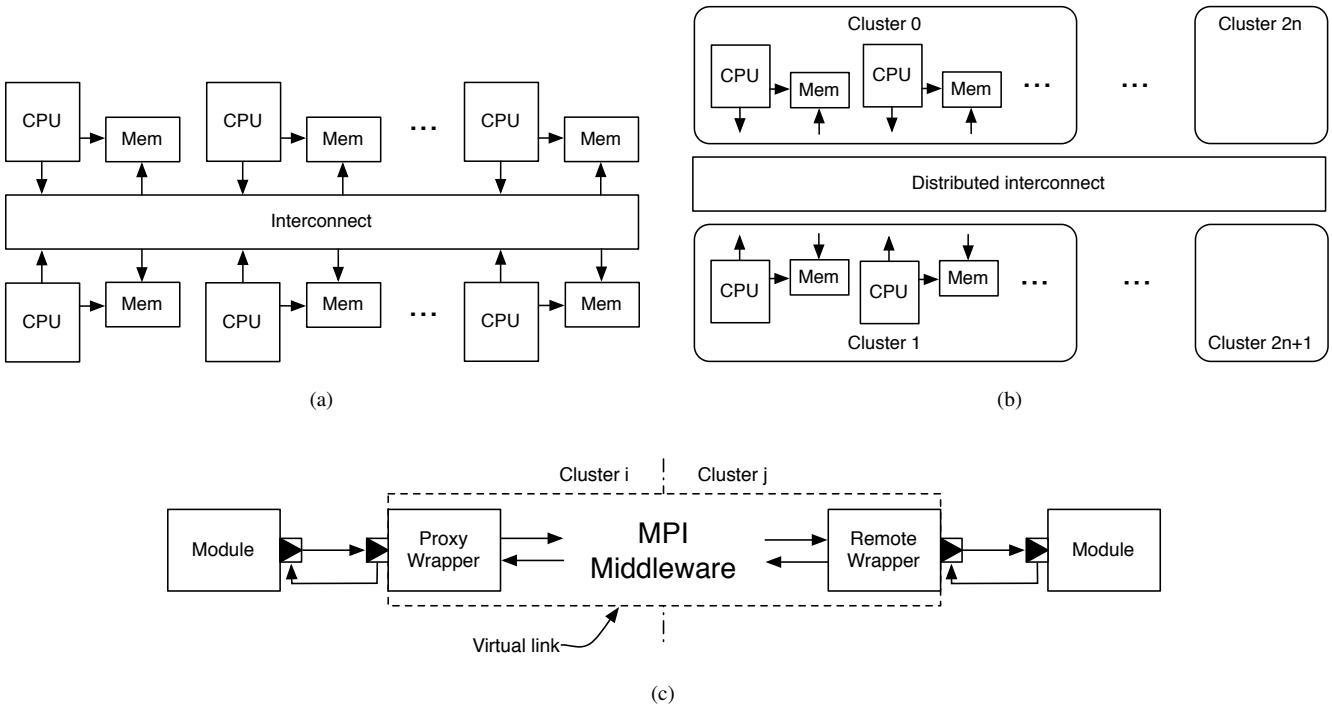

Fig. 2. Using our approach, the parallelization of a SystemC simulation begins with the transformation of the non-distributed model (a) of the simulated system to a distributed one (b). In the same time, the interconnect is split across clusters and, thereafter, is known as *distributed interconnect*. Finally, links carrying communication through the interconnect are replaced by virtual links (c). The latter exchange data through the MPI middleware abstracted and hidden in the distributed interconnect.

diverging in time with one another. This part of the whole synchronization approach implements a handshake between clusters sharing at least one virtual link. The implementation takes place in a SystemC module named *synchronization module*, one assigned to each cluster.

In the following paragraphs, we will deeper detail wrapping and synchronization mechanisms and implementations.

A. Wrappers

In a standard non-distributed SystemC TLM simulation, an initiator communicates with its targets through *transport* methods, where the name *method* refers to a C++ one. A transport method literally transports a transaction from an initiator to a target, or at least to an intermediate SystemC module acting like a target when modules are chained. When the simulation is distributed, an initiator and a target communicating together may be mapped onto different clusters. So as to hide this mapping from both initiator and target point of views, wrappers are introduced in the simulated system model. These wrappers transform the call to the transport method into an asynchronous MPI communication, thereby implementing the abstraction exposed by the virtual link.

On the initiator side, the wrapper is named the *proxy wrapper*; on the target side, the wrapper is named the *remote wrapper*. The proxy wrapper mimics the behavior of the target as if the initiator was directly connected to it. The remote wrapper is in charge of forwarding the request to the target and the response back to the initiator. In order to handle the asynchronous nature of MPI communication, both wrappers are implemented as SystemC modules. They contain one process that is activated when a transaction is received. This activation is managed by an *event dispatcher*, one assigned to each cluster. The event dispatcher polls MPI communication from remote clusters sharing at least one virtual link with the cluster owning the event dispatcher. Figure 3 illustrates communication between an initiator and a target.

Wrapper association, used to create a virtual link, is made before the simulation begins. Associations are specified in a configuration file, containing wrappers unique identifiers. This file is loaded at start-up and parsed to build virtual links.

B. Synchronization module

Synchronization in a parallel simulation aims to keep the simulation state consistent among all parts of the simulation environment. In our case, synchronization only occurs between clusters sharing at least one communication dependency between an initiator and a target, assuming both modules are on different clusters. Consequently, if two clusters do not share such dependency, they never synchronize.

The synchronization module implements the explicit part of our hybrid synchronization approach as a handshake between clusters. The implementation of this handshake is detailed in figure 4. In opposite to distributed transactional communication, synchronization is done synchronously. Then, the main issue is to prevent deadlocks as SystemC processes are

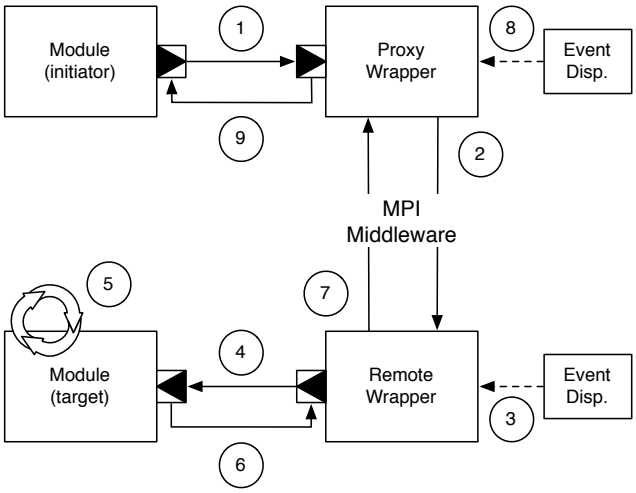

Fig. 3. When an initiator sends a transaction to a target, the proxy wrapper connected to this initiator receives the transaction (1). The proxy wrapper forwards the transaction to its associated remote wrapper (2) and waits for the response. On the other side, the remote wrapper wakes up when the transaction arrives (3), previously notified by the event dispatcher of its cluster. The remote wrapper transfers the transaction to the target (4), which processes it (5). When the target returns the response (6), the remote wrapper sends it back to the proxy wrapper (7), which is notified by the initiator's cluster event dispatcher (8). Finally, the proxy wrapper forwards the response to the initiator (9).

evaluated sequentially. To do so, explicit synchronization is done as follows, considering the point of view of a cluster:

1) A synchronization order is sent (`send_sync`) to all clusters containing one or more targets connected to one or more initiator in the current cluster (lines 2-4);
2) Upon a synchronization order is received from an initiator's cluster (`recv_sync`), an acknowledge is sent back (`send_ack`) to it (lines 5-8);
3) Finally, the current cluster wait (`recv_ack`) for receiving all acknowledges for all synchronization orders it sent at the first phase (lines 9-11).

Explicit synchronization occurs at regular intervals in the simulation. The time spent during two explicit synchronizations is called the *synchronization period*. This period is specified by the simulated system designer and/or simulation end-user. So as to reduce the cost of synchronization during the simulation, explicit synchronization is implemented as a SystemC *method process*. This kind of SystemC process (i.e `SC_METHOD`) corresponds to a function call, where the other kind, named a *thread process* (i.e. `SC_THREAD`), generates thread context switches, which is more expensive.

Nonetheless, it is mandatory to guarantee that explicit synchronization will effectively bound the temporal error introduced by asynchronous communication between clusters. Actually, this is already guaranteed by the nature of explicit synchronization and its periodicity. Indeed, explicit synchronization occurs in each cluster with the same period and the synchronization is processed synchronously between related clusters. Moreover, all clusters start their part of the simulation

```
1:  procedure EXPLICIT_SYNCHRONIZATION
2:      for c_t ∈ remote_target_clusters do
3:          SEND_SYNC(c_t)
4:      end for
5:      for c_i ∈ remote_initiator_clusters do
6:          RECV_SYNC(c_i)
7:          SEND_ACK(c_i)
8:      end for
9:      for c_t ∈ remote_target_clusters do
10:         RECV_ACK(c_t)
11:     end for
12: end procedure
```

Fig. 4. Listing of the explicit synchronization algorithm.

evaluation at time zero. As a result, when clusters synchronize, their local times converge to a global simulation time.

Figure 5 illustrates how explicit synchronization effectively bounds the temporal error. In the given example, at step (2), cluster j is in advance compared to cluster i. Their local times differ by a certain δ, previously defined as the temporal error. Next, at step (3), cluster i is in advance compared to cluster j and their local times differ again by a certain, possibly different, δ. Finally, at step (4), clusters initiate a handshake and block until the synchronization completes. Thereafter, cluster i and j continue the simulation with their synchronized local times.

Simulation termination is another issue we address in this paper. When a part of the whole simulation terminates in a cluster, there is, a priori, no reason for other clusters to know whether that cluster has finished its part of the simulation. Worse, some clusters may wait for the terminated cluster to synchronize, resulting in a deadlock.

So as to deal with this issue, we propose to use a cooperative termination method, based on the algorithm of explicit synchronization (figure 4). When the simulation is expected to end, a call to the `sim_stop` function is made. This causes synchronization modules to send *stop orders* instead of synchronization one. Then, all clusters are allowed to respond to pending transactions until the last (stop) synchronization is achieved.

IV. Validation environment

The validation environment is composed of two parts: a dedicated hardware simulation infrastructure and a SystemC TLM validation model.

A. Simulation hardware

The hardware used for validation is composed of four nodes. Each node is a quad-core Xeon W3550 at 3.06 GHz with 24 GB RAM and two 1000 BaseT network interfaces. All run a 2.6.9-67 RHEL 4 SMP Linux kernel without support for HyperThreading.

The MPI implementation is OpenMPI version 1.4.2. The command line used to launch distributed simulation depends on the number of clusters involved in the simulation.

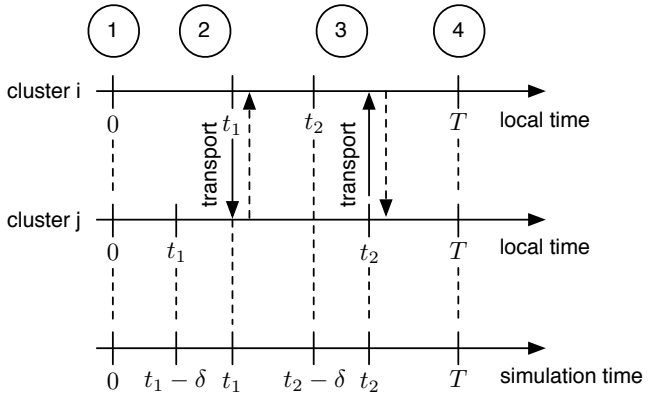

Fig. 5. Example of relative time deviation between clusters and effect of explicit synchronization on the local time of clusters. First, all clusters start the simulation at time zero (1). Next, a cluster i's initiator sends a transaction to a target of cluster j through a transport method and receives a response (2). After a little while, a cluster j's initiator send a transaction to a target of cluster i and receives a response (3). When the synchronization period (T) has elapsed, cluster i and cluster j synchronize synchronously (4).

For instance, when using 4 clusters, the command line looks like `mpiexec --mca btl tcp,self --mca mpi_paffinity_alone 1 -hostfile mpi.hosts -n 4 ./top`.

B. Validation model

The hardware part of the validation model (figure 2a) is composed of processing cores (CPU's), distributed memories (Mem) and a network interconnect. Each memory is accessible directly to one processor as its local memory, using a dedicated port, and to all other processors through the network interconnect. The address space is globally shared. Communication between CPU's is made thanks through local memories and is assumed to be unpredictable.

The application implemented for validation purpose follows a data-flow software model. In this context, each CPU executes two tasks. The first one aims to represent realistic thread context switches involved in genuine complex system simulations. It implements a fixed-length integer sum. This task is executed at each cycle and does not produce communications outside the CPU. The second task can be a producer or a consumer task, implementing a variable length integer sum. This task is executed periodically following a user-defined simulated period (given in cycles). In a producer task, the sum is processed and the result is written to a consumer local memory. In a consumer task, the sum is processed as soon as a new data is available in its local memory. This new data is used as the initial value of the processed sum. For the purpose of the validation, producers and consumers are chained two-by-two, each producer providing data to only one consumer.

In order to distribute the SystemC model of figure 2a, the model is cut up into clusters as defined in section II. In the present case, CPU's are grouped with their local memories and the network interconnect is distributed among clusters. The resulting SystemC model is presented figure 2b. The transformation from a standard non-distributed SystemC model to a distributed one is completely automated. Though, the current implementation of this automated transformation is only valid for the module pattern of figure 2a at this time, the transformation process can be easily extended to a general case. For instance, SystemCXML [10], PinaVM [11] and Scoot [12] are tools that can be used to extract communication dependency information and generate a top-level SystemC module with the appropriate allocation of clusters.

The validation model we propose here is customizable. The following parameters can be changed: the number of CPU's, the computational load in CPU's, the local memory size, the number of clusters and the explicit synchronization period. Nevertheless, in order to keep the testing set size reasonable, some constraints have been put on the model parameter values: producer and consumer tasks have all the same computational load (i.e. the same sum size) and the explicit synchronization period is identical for all clusters.

It is important to notice that the system model used for validation only aims to be theoretical and does not intend to be implemented on silicon. Nevertheless, it aims to express the following system properties: a high degree of parallelism through a great number of tasks and a high degree of interconnection.

V. Experimental results

For each distributed simulation, the validation model is composed of 64 CPU's and 64 memories. The distribution is made following three configurations for which the simulated system model is partitioned in 4, 8 and 16 clusters respectively. The number of modules per cluster is chosen so as to balance the load among node resources. In addition, we forced a cluster to be evaluated alone on one node core.

In order to characterize the behavior of a distributed SystemC TLM simulation using our approach, we ran the simulations in three modes, corresponding to three producer/consumer task simulated period values:

- using a fixed value of 100 cycles (no random variation);
- following an uniform law of probability with a mean of 100 cycles and a variance of 20% (i.e. 20 cycles);
- following a Poisson law of probability with a mean of 100 cycles and a variance of 20% (i.e. 20 cycles).

We also ran distributed simulations with a variance equals to 40% of the mean. In those cases, the results were similar to ones exposed thereafter, so we will not discuss them in this paper.

Figure 6 shows the results we obtained for each distributed configuration and each producer/consumer task simulated period value. The results present the influence of the real-time duration of producer/consumer tasks for a given simulated period, which we are going to discuss now.

A. Speed-up characterization

Figure 6a, 6d and 6g show up the relationship between T_c, the real-time producer/consumer task duration, and T_{es}, the explicit synchronization period. One can notice that T_c

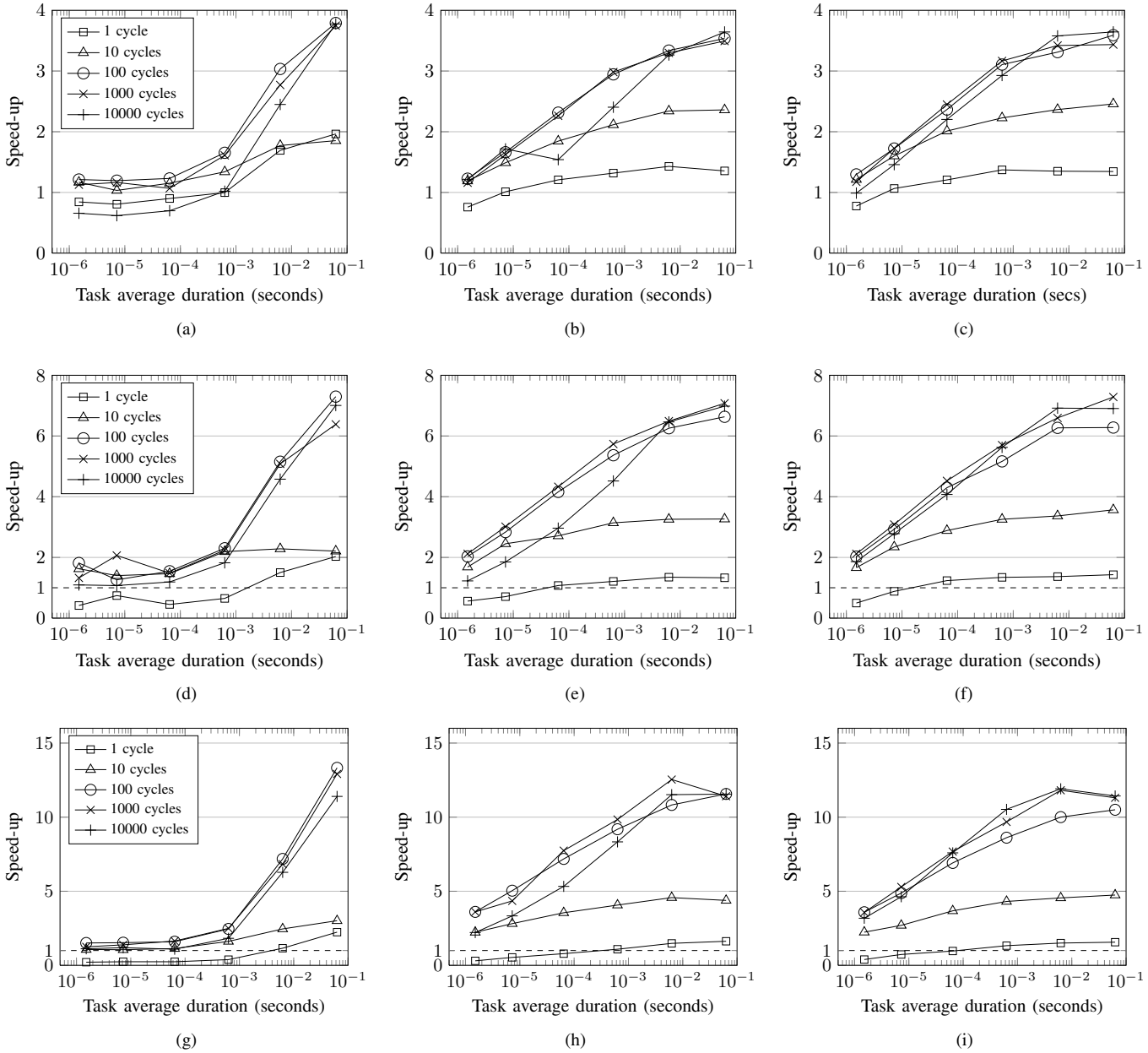

Fig. 6. Speed-up of SystemC TLM distributed simulations compared to a non-distributed one. These results present the speed-up we achieved using our new approach, considering several explicit synchronization periods given in cycles. Rows correspond, in order, to the three distributed configuration composed of 4, 8, 16 clusters. The first column gives the results for a fixed value of the simulated producer/consumer task period. The second and third columns give the results for a random producer/consumer task simulated period, following an uniform probability law and a Poisson probability law respectively.

also represents the communication real-time period. When $T_{es} < T_c$, more than one synchronization occur during a period of T_c. As a consequence, the number of thread context switches grows within the SystemC simulation engine, slowing down the distributed simulation. On the other side, increasing T_{es} beyond T_c does not provide a significant benefit. Indeed, simulation throughput is limited by implicit synchronization, which its period equals T_c.

As shown in the second and third columns of figure 6, a random producer/consumer task simulated period gives a better speed-up for shorter values of T_c than with a non-random simulated task period. This is explained by the probability that the producer/consumer task simulated period is less than 100 cycles (i.e. the mean value). Therefore, the number of synchronization during a period of T_c decreases. In addition, the acceleration given by such a scenario is greater than the slowdown caused when the task simulated period value is greater than 100 cycles.

Looking at validation results, our approach is scalable. Indeed, the maximum speed-up in all cases is very close to the theoretical speed-up. For instance, for a non-random SystemC TLM distributed simulation composed of 16 clusters,

TABLE I
TEMPORAL ERROR GIVEN IN PERCENT OF THE PROCUDER/CONSUMER TASK SIMULATED PERIOD (100 CYCLES). VALUES GIVEN HERE ARE THE MAXIMUM GENERATED ERROR CONSIDERING ALL VALUES OF T_c.

# clusters	Explicit synchronization period				
	1	10	100	1000	10 000
4	1.00%	4.2%	8.6%	143.90%	1517.70%
8	0.99%	3.4%	13.1%	231.90%	1388.60%
16	0.95%	4.2%	15.5%	73.00%	2193.60%

the speed-up equals 13.33 compared to a non-distributed simulation; the maximum speed-up for distributed simulation composed of 16 clusters and following a uniform random law equals 12.54.

B. Error characterization

Table I shows the temporal error, as defined in section II, along with different values of the explicit synchronization period.

One can see that the temporal error is always bounded by the explicit synchronization period. For instance, given an explicit synchronization period of 10 cycles, the maximum error equals 4.2% in average. As expected, when the explicit synchronization period gets longer, the temporal error increases. In addition, as detailed in section IV, communication in producer and consumer tasks is surrounded by task evaluations. In the case of a SystemC TLM simulation, these tasks are sequentially evaluated like any other concurrent SystemC process in a given simulation engine. Then, when a producer/consumer task is evaluated, clusters do not synchronize with one another. Therefore, the longer the producer/consumer task evaluation, the greater the temporal error.

C. Exploiting distributed simulation properties

When looking at the boundary behavior, two distinct situations can be observed according to the explicit synchronization period. Indeed, on one hand, short periods ($<$ 10 cycles) give little throughput but high precision. On the other hand, long periods ($>$ 100 cycles) give higher throughput, but little precision. Therefore, we propose two simulations modes: one ought to use a period around 100 cycles when expecting a high throughput; in return one ought to use an explicit synchronization period of 1 cycle when more precision is required.

VI. RELATED WORK

SystemC is a discrete event simulator using *delta cycles* to simulate concurrent processes in a system. Such a process can be modeled like a function call or a thread depending on its nature. Buchmann [13], Mouchard [14] and Naguib [15] observed that the default SystemC dynamic process scheduling produces more thread context switches than effectively needed. So, they proposed a static scheduling relying on communication dependencies between SystemC processes. The scheduling is obtained thanks to a static analysis of the simulated system model. However, when communication is unpredictable this approach cannot be used. An alternative is to parallelize concurrent process evaluation. To do so, two methods are exposed in related work.

One method requires modifying the SystemC simulation engine. Ezudheen [1] do it by adding OpenMP [16] directives while Mello [2] use the QuickThread framework [17]. More radically, Nanjundappa [18] implement a transformation chain from SystemC to CUDA [19]. All these proposals make severe modification of the SystemC implementation that leads to relevant results. However, such modification implies an expensive maintenance to stay compatible with future versions of SystemC. Our aim is to focus on the synchronization mechanism being one of the most critical part in parallel simulation. The other parallelization method is the one our approach relies on and we detailed in section II.

Combes [3] show up the synchronization bottleneck generated by the conservative variant of PDES. As a solution, they propose an interesting distributed synchronization mechanism. However, it requires modifying the SystemC simulation engine what we proscribed. Our synchronization implementation is close to their approach but ours is much simpler. We implement it at model level instead of inside the SystemC simulation engine.

Yi [20] proposes an interesting method, called *trace-driven virtual synchronization*, that separates event generation from time alignment. However, implementing this in the context of a SystemC simulation requires modifying the simulation engine, which we exclude in this paper.

VII. CONCLUSION

In conclusion, this paper presents a new parallel approach with a hybrid synchronization mechanism designed to deal with unpredictable systems, offering to simulate and explore design space of such systems. Simulation parallelization is transparent to the simulated system designer thanks to a dedicated SystemC TLM framework, thereby increasing reuse of previously written SystemC model. This framework does not require modifying the SystemC simulation engine at all. Both features provide an easy to use and relevant simulation environment.

Experimental results show that the distributed simulation speed-up is conditioned by a threshold. This threshold is inherent to distributed programming methods and relies on the ratio between the simulation processing and the communication cost. Results also highlight the relationship between the synchronization mechanism and the nature of communication between clusters. Two simulation modes can be extracted from results. A first one that provides little throughput but high precision. In return, the second mode provides high throughput but little precision.

In this second case, results underline that when the communication simulated period is fixed, corresponding to the ideal case, the best speed-up is obtained when the explicit synchronization period overlaps with the communication period. In addition, when the communication simulated period is randomized, approaching the real case, acceleration values are a few smaller but still brasatisfying. For instance, the speed-up

with a distributed simulation composed of 16 clusters compared to a non-distributed simulation equals 13.33 and 12.54 for non-random and random cases respectively. In all cases, the temporal error is bounded by explicit synchronization period. These results require comparison with those of distributed simulations using more realistic SystemC TLM models to be fully validated.

REFERENCES

[1] P. Ezudheen, P. Chandran, J. Chandra, B.P. Simon, and D. Ravi, "Parallelizing SystemC Kernel for Fast Hardware Simulation on SMP Machines," in *Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*. Montreal, Canada, pp. 80–87, 2009.

[2] A. Mello, I. Maia, A. Greiner, and F. Pecheux, "Parallel Simulation of SystemC TLM 2.0 Compliant MPSoC on SMP Workstations," in *Proceedings of Design, Automation and Test in Europe (DATE)*, Dresden, Germany, pp. 606–609, 2010.

[3] P. Combes, E. Caron, P. Desprez, B. Chopard and J. Zory, "Relaxing Synchronization in a Parallel SystemC Kernel," in *Proceedings of IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, Sydney, Australia, pp. 180–187, 2008.

[4] B. Chopard, P. Combes, and J. Zory, "A Conservative Approach to SystemC Parallelization," in *Proceedings of International Conference on Computational Science (ICCS)*, Reading, United Kingdom, pp. 653–660, 2006.

[5] M. Trams, "Conservative Distributed Discrete Event Simulation with SystemC using Explicit Lookahead," *Digital Force White Papers*, 2004.

[6] V. Galiano, H. Migallón, D. Pérez-Caparrós, M. Martínez, "Distributing SystemC Structures in Parallel Simulations," in *Proceedings of the 2009 Spring Simulation Multiconference*, San Diego, CA, United States, pp. 1–8, 2009.

[7] D. R. Cox, "RITSim: Distributed SystemC Simulation," *Master thesis* at Kate Gleason College of Engineering, 2005.

[8] R. M. Fujimoto, "Parallel Discrete Event Simulation," in *Proceedings of the 21st Conference on Winter Simulation*,Washington D.C., United States, pp. 19–28, 1989.

[9] Message Passing Interface Forum, "MPI: a Message Passing Interface Standard," Stuttgart, Germany, 2009.

[10] D. Berner, J.-P. Talpin, H. D. Patel, D. Mathaikutty, S. K. Shukla. "SystemCXML: An Exstensible SystemC Front-end Using XML," in *Proceedings of Forum on specification and Design Languages (FDL)*, Lausanne, Switzerland, pp. 405–409, 2005.

[11] K. Marquet, M. Moy, "PinaVM: a SystemC Front-end Based on an Executable Intermediate Representation," in *Proceedings of the 10th ACM International Conference on Embedded Software (ICES)*, Scottsdale, AZ, United States, pp. 79–88, 2010.

[12] N. Blanc, D. Kroening, N. Sharygina, "Scoot: A Tool for the Analysis of SystemC Models," in *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Budapest, Hungary, 2008, pp. 467–470.

[13] R. Buchmann and A. Greiner, "A Fully Static Scheduling Approach for Fast Cycle Accurate SystemC Simulation of MPSoCs," in *Proceedings of International Conference on Microelectronics (ICM)*, Cairo, Egypt, pp. 105–108, 2007.

[14] G. Mouchard, D. G. Pérez, and O. Temam, "FastSysC: A Fast Simulation Engine," in *Proceedings of Design, Automation and Test in Europe (DATE)*, Paris, France, 2004.

[15] Y. N. Naguib and R. S. Guindi, "Speeding up SystemC Simulation Through Process Splitting," in *Proceedings of Design, Automation and Test in Europe (DATE)*, Nice, France, pp. 111–116, 2007.

[16] OpenMP Architecture Review Board, "OpenMP: The OpenMP API specification for parallel programming," http://www.openmp.org.

[17] QuickThread Programming, LLC, "QuickThread framework,", http://www.quickthreadprogrammin.com

[18] M. Nanjundappa, H. D. Patel, B. A. Jose, and S. K. Shukla, "SCGPSim: A Fast SystemC Simulator on GPUs," in *Proceedings of the 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*, Taipei, Taiwan, pp. 149–154, 2010.

[19] NVidia, "CUDA technology," http://www.nvidia.com.

[20] Y. Yi, D. Kim, S. Ha, "Fast and Accurate Cosimulation of MPSoC Using Trace-Driven Virtual Synchronization," in *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*,Vol. 26, No. 12, pp. 2186–2200, 2007.