# Scaling FMI-CS Based Multi-Simulation Beyond Thousand FMUs on Infiniband Cluster

Stephane Vialle[1,2] Jean-Philippe Tavella[3] Cherifa Dad[1] Remi Corniglion[3] Mathieu Caujolle[3] Vincent Reinbold [4]

[1]UMI 2958 - GT-CNRS, CentraleSupelec, Université Paris-Saclay, 57070 Metz, France
[2]LRI - UMR 8623, 91190 Gif-sur-Yvette, France
[3]EDF Lab Saclay, 91120 Palaiseau, France
[4]University of Leuven, Department of Civil Engineering, 3001 Leuven, Belgium

## Abstract

In recent years, co-simulation has become an increasingly industrial tool to simulate Cyber Physical Systems including multi-physics and control, like *smart electric grids*, since it allows to involve different modeling tools within the same temporal simulation. The challenge now is to integrate in a single calculation scheme very numerous and intensely inter-connected models, and to do it without any loss in model accuracy. This will avoid neglecting fine phenomena or moving away from the basic principle of equation-based modeling.

Offering both a large number of computing cores and a large amount of distributed memory, multi-core PC clusters can address this key issue in order to achieve huge multi-simulations in acceptable time. This paper introduces all our efforts to parallelize and distribute our co-simulation environment based on the FMI for Co-Simulation standard (FMI-CS). At the end of 2016 we succeeded to scale beyond 1000 FMUs and 1000 computing cores on different PC-clusters, including the most recent HPC Infiniband-cluster available at EDF.

*Keywords: Multi-Simulation, FMI, Scaling, Multi-core, PC Cluster*

## 1 Introduction and Objectives

A multi-simulation based on the FMI for Co-Simulation standard is a graph of communicating components (named FMUs) achieving a time stepped integration, under supervision of a global control unit (named *Master Algorithm* in the standard), as illustrated in Fig. 1. During a time step, all FMU inputs remain constant and all FMUs can be run concurrently. When all FMU computations of a time step are finished, the FMU outputs are routed to connected FMU inputs, and all FMUs communicate with the Master Algorithm. When using adaptive time steps or managing events inside the time steps, the Master Algorithm has a complex role to decide on the next time step to execute.

In order to run wide multi-simulations requiring large memory and heavy CPU resource consumption, we first need to distribute and process a co-simulation graph on several PCs. Second, to achieve *scaling* we need (1) to speedup a co-simulation using more computing resources (cores and PCs), and (2) to strive to maintain the same execution time when running larger co-simulations on more

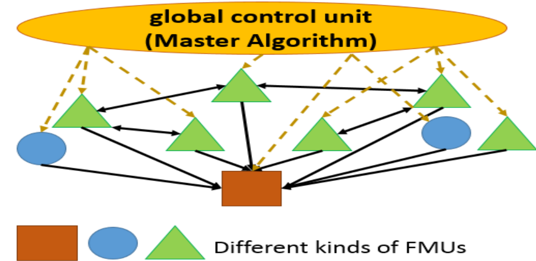


**Figure 1.** Generic FMU graph implementing a multi-simulation

PCs. Of course to achieve this, we attempt to minimize the global execution time as the sum of all the parallel FMU computation substeps, the FMU communication substeps, and the graph control substeps.

In 2014 we designed a distributed and parallel FMI based multi-simulation environment, named DACCOSIM[1] (Galtier et al., 2015), integrating a hierarchical and distributed Master Algorithm. Available under AGPL for both Windows and Linux operating systems, DACCOSIM[2] achieves a multi-threaded execution of local FMUs on each node with concurrent run of different FMUs on cluster nodes, and frequent data exchange between nodes (see section 2). Then, we decided to optimize and facilitate the execution of our environment on multi-core PC clusters, which are our typical computing platforms. Unfortunately, FMUs are kind of opaque computing tasks frequently exchanging small messages. They are very different from optimized High Performance Computing tasks, and we faced different difficulties. In 2016 we succeeded to exhibit scaling on a co-simulation of heat transfers in a set of $n$ three-floor buildings, and we efficiently run up to 81 FMUs on a 12-node PC cluster with a 10 Gbit/s Ethernet interconnect and 6 cores per node (Dad et al., 2016). However, we needed to identify the optimal distribution of one building on a minimum set of cluster nodes after running several benchmarks, and we scaled our co-simulation replicating our initial building and its optimal distribution. This approach has proven it is possible to achieve scaling on distributed FMI based co-simulations, despite the unusual features of an FMU task graph, from a classic parallel computing point of view. We have carried on with this work, in order to automate the ef-

[1]`https://daccosim.foundry.supelec.fr`
[2]Partially supported by Region Lorraine (France)

ficient distribution of wide co-simulations on large multi-core PC clusters, aiming to distributed thousands of FMUs on thousands of computing cores.

The paper is organized as follow. Next section describes the features of an FMU graph execution, the principles of its execution, and the choices done when designing the DACCOSIM architecture. Section 3 lists all sources of parallelism and also performance losses in an FMU graph running on a multi-core PC cluster, in order to design an efficient software architecture of multi-simulation. Section 4 investigates the distribution of the FMU graph on a multi-core PC cluster, with poor or rich meta-data on the FMU graph, in order to maximize performance. Then, section 5 introduces our new large scale benchmark detailing reached numerical results, performance and scalability. Finally, section 6 lists our current results and remaining challenges.

# 2 FMI-CS based Multi-Simulations

## 2.1 FMI-CS Strengths and Limitations

Modern electric systems are made of numerous interacting subsystems: power grid, automated meter management, centralized and decentralized production, demand side management (including smart charging for electric vehicle), storage, ICT resources... Beyond a consensus on the language to use, modeling wide and complex systems in one universal modeling tool implies to make some simplifications that may lead to minimize important phenomena. As historic and domain-specific tools validated their business libraries since a long time, the most rational approach to simulate wide Cyber Physical Systems (CPSs) consists in recycling specialized simulation tools in a co-simulation approach.

The Functional Mock-up Interface for Co-Simulation (FMI-CS) specification can now be considered as a well-established standard for co-simulation thanks to numerous developments done by industrial parties (Blochwitz et al., 2011). A growing number of business tools - like EMTP-RV[3] for electromagnetic transient modeling - have adopted the standard and added FMI connectors to their products. FMI-CS allows to obtain a fairly realistic representation of the whole system behavior since all the subsystems are equally taken into account without the pre-eminence of a domain (e.g. ICT) on another (e.g. physics). It allows the building of stand-alone active components (FMUs) that can be executed independently of each other. FMUs exchange data (with other FMUs or with external components) only at some discrete communication points. In the time interval between two communication points each FMU model is simulated by its own numerical solver, and a *Master Algorithm* controls the FMU graph at each communication point (see Fig. 1).

An FMU for Co-Simulation consists of a ZIP file containing an XML-based model description and a dynamic

[3]http://emtp-software.com/

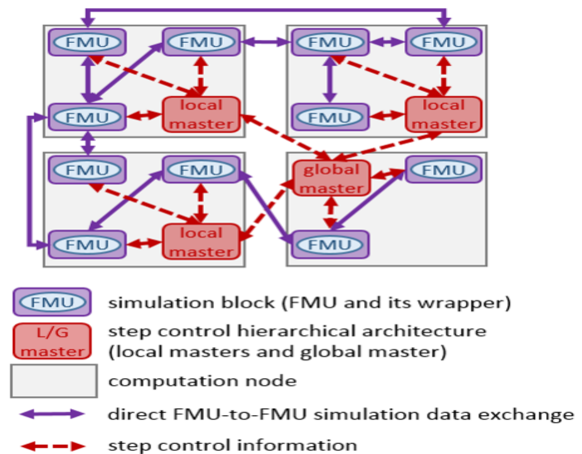


**Figure 2.** DACCOSIM software architecture

library being either a self-contained executable component or a call to a third-party tool at run-time (tool coupling). FMI-CS is focused on the slave side (FMU) and remains very discreet on the master side (Master Algorithm). The way a simulation tool utilizes the functionality provided by FMUs is strongly related to the simulation paradigm on which the tool relies:

- Some co-simulation middleware use the Agent & Artifact paradigm (Ricci et al., 2007) to describe an heterogeneous multi-model, and they rely on the Discrete EVent System Specification (DEVS) formalism (Zeigler et al., 2000) to conceive a decentralized execution algorithm respecting causality constraints. But conservative algorithms, such as Chandy-Misra-Bryant (Chandy and Misra, 1979) used in some A&A tools like MECSYCO[4], do not integrate the concept of rollback. They must be adapted to restore FMUs to a previous state and adjust the step size in case of events or fast system dynamics (Camus et al., 2016).
- Another class of tools is based on the synchronization of the communication points of all the FMUs involved in a calculation graph. Unfortunately these tools often stick to the master pseudo codes given as examples in the FMI standard with a centralized algorithm acting as a bottleneck for all the communication (data exchanges and control information).

## 2.2 DACCOSIM Architecture Choices

In our attempts to design, distribute and co-simulate on cluster nodes very wide systems composed of thousands of FMUs, we need both to synchronize all the communication points (for accuracy purpose) and decentralize the usual control function of the Master Algorithm (for performance purpose). First versions of these features were available within DACCOSIM in 2014.

DACCOSIM 2017 emphasizes a complete and user-friendly Graphical User Interface (GUI) to configure and perform local or distributed co-simulations with potentially many heterogeneous FMUs compliant with the co-

[4]http://mecsyco.com/

simulation part of the FMI 2.0 standard (FMI-CS 2.0). A DACCOSIM calculation graph consists of blocks (mainly FMUs) that are connected by data-flow links and potentially distributed on different computation nodes. The graph is then translated into Java master codes in conformance with the features described in the FMI-CS 2.0 standard. More precisely, DACCOSIM notably offers for the co-initialization of its calculation graph:

- Automatic construction of the global causal dependency graph, built both from the FMUs internal dependencies and the calculation graph external dependencies. An acyclic view of the graph is generated by aggregating each cycle as a super-node composed of Strongly Connected Components (SCCs);
- Generalized distributed co-initialization algorithm, mixing a sequential propagation method applied to the acyclic dependency graph, and a Newton-Raphson method solving its SCCs.

And for co-simulation, it offers among others:

- Implementation of each FMU wrapper as two threads allowing to concurrently run computations and send messages (FMU & control) while receiving incoming messages;
- Overlapped (optimistic) or ordered (pessimistic) data synchronization inside distributed masters (see section 3.3), that can operate with constant time steps or with adaptive time steps controlled by one-step methods (Euler or Richardson) or a multi-step method (Adams-Bashforth);
- Approximate event detection while waiting for a new version of the FMI standard able to correctly handle hybrid co-simulations (Tavella et al., 2016).

DACCOSIM generated master codes follow a centralized hierarchical approach (see Fig. 2). A unique global master located on one cluster node is in charge of handling the control data coming from several local masters located on other cluster nodes and taking step by step decisions based on this information. Every master, whether global or local, aggregates these control data that are coming from each FMU wrapper present on its cluster node. This is done before communicating synthesized information to the global master. The control data exchanged between masters and between FMUs and masters are called vertical data. Of course when the co-simulation is run on a single machine, only one master code is generated.

An original feature of the DACCOSIM architecture lies in the fact that the FMU variable values to be exchanged at each communication step are directly transmitted from the senders to receivers without passing by a master. The masters, the wrapped blocks (mainly FMUs with wrappers) as well as the communication channels between them are automatically generated by DACCOSIM by translating the calculation graph defined by the user via its GUI. All communications are implemented using ZeroMQ (or ZMQ) middleware, allowing direct communications between different threads located on the same or on different PC cluster nodes. For intra-node communications, a mechanism of shared message queue is also available.

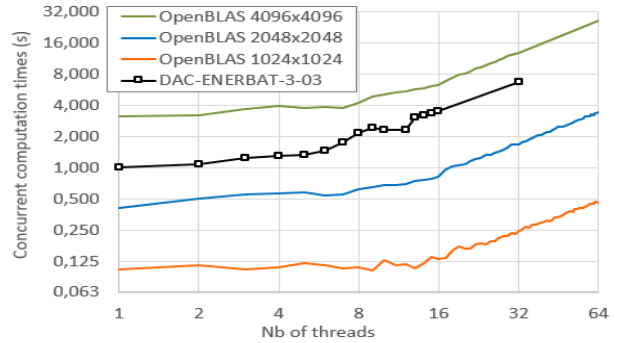

**Figure 3.** Concurrent run times on a 2x4-core node

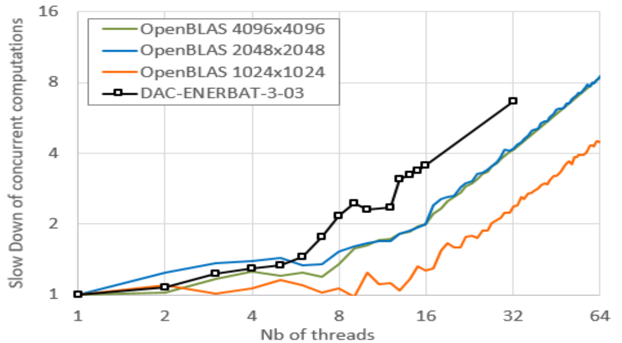

**Figure 4.** Slow down of concurrent runs on a 2x4-core node

# 3 Parallelism Sources and Limitations

## 3.1 FMU Computations

Each computing substep is a high source of parallelism, as all FMUs can concurrently achieve their computations. However, running $n_f$ FMUs on $n_c$ cores of the same computing node can lead to imperfect concurrency: (1) when there are less cores than FMUs ($n_c < n_f$), and (2) when the FMU computations access the node memory and saturate the memory bandwith. Taking into account this FMU concurrency imperfection, we will deduce the optimal number of FMUs to run on each computing node, and so the total number of nodes to use (see section 4).

### 3.1.1 FMU Concurrency Experiment

We concurrently run HPC computing kernels of *dense matrix product* ($C = A \times B$, a reference HPC benchmark) on one of our cluster computing node. We used an *OpenBLAS dgemm* kernel, and a NUMA dual-haswell node with $2 \times 4$ physical cores at 3.5 GHz, and $2 \times 15$ MB of cache memory. Fig. 3 shows the execution times measured on different problem sizes, with optimized *OpenBLAS* kernels blocking data in cache to minimize the memory bandwidth consumption. For each problem size, we concurrently run from 1 up to 64 threads, each thread executing one complete call to the kernel on locally allocated data structures. We considered (1) two large matrix sizes: $2048 \times 2048$ matrices of 32 MB and $4096 \times 4096$ matrices of 128 MB,

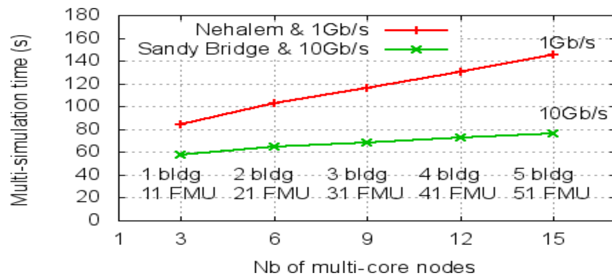


**Figure 5.** Size up experiments on 1 Gb/s and 10 Gb/s clusters

each matrix being larger than the entire node cache, and (2) a smaller problem with $1024 \times 1024$ matrices of 8 MB, allowing to store the three matrices of the problem into one node cache. Each curve illustrates the global execution time evolution when running more concurrent computations.

Fig. 4 shows the slow down observed when increasing the number of concurrent computations: $SD(n) = t(n)/t(1)$. Our optimized OpenBLAS kernel exhibits good concurrent performance, with a very limited slow down up to the 8 physical cores, followed by a first slow down increase up to the 16 logical cores (exploiting hyper-threading), and a linear increase when running more concurrent tasks serially processed by the node cores. Then, we concurrently run several threads executing the same FMU[5], modeling heat transfers and achieving significant computations with the *cvode* solver[6]. We can observe in Fig. 4 that these concurrent FMU executions (1) exhibit a limited slow down, up to $(n_c - 2)$ FMU computing threads, similar to the behavior of concurrent OpenBLAS kernels, and (2) then quickly increase their slow down beyond $(n_c - 2)$ FMU computing threads per node, going away from OpenBLAS kernel curves.

In fact some extra tasks are running in parallel of the FMU computation threads in DACCOSIM, and it is not surprising the slow down starts to increase a little bit before deploying one FMU per physical core. But the slow down increase appears stronger than with OpenBLAS kernels, and is militantly in favour of running only $(n_c - 2)$ FMUs per computing node and using additive nodes. Of course, this experimental study will need to be conducted on different cluster nodes with different FMUs in the near future to confirm the definition of the ideal number of FMUs to deploy and run on a multi-core cluster node.

#### 3.1.2 Unsuccessful Performance Improvement

When the number of available computing nodes is limited, it leads to run many FMUs on a same node, many more than $(n_c - 2)$. Then, we attempted to reduce the computation time limiting the number of FMUs simultaneously run in parallel on a same computing node. We implemented a semaphore-based synchronization-mechanism, authorizing only $n_{max}$ FMUs to concurrently run their computations (a new FMU could enter its computation substep only when a previous one finished its substep).

But performance measured when limiting the concurrency of many FMUs on a same node were disappointing: the total computation time still increased. We have not succeeded to improve the execution of many concurrent FMUs per node with a basic scheduling mechanism.

### 3.2 FMU Communications

#### 3.2.1 Main Features of Inter-FMU Communications

There is no order in the inter-FMU communications of a time step, they can all be routed in parallel, fully exploiting the cluster interconnect bandwidth. Moreover, FMU communications inside a computing node can be achieved faster (no crossing of network connections no network software layer). But data exchanged between two FMUs are usually small (like one or a few floating point values). Each FMU communication is sensitive to the network and applicative latency: time to transfer a byte from one JVM (running FMUs) on one node to another JVM on another node, in current DACCOSIM implementation. Moreover, an FMU graph has many connections generating communications at the end of each time step.

So, communication features of multi-simulations are different from classic HPC application ones, which always attempt to group data and exchange large messages not too frequently. FMU communications are small, numerous and frequent, however their implementation can be parallelized.

#### 3.2.2 Sensitivity to Latency and Bandwidth

Respective weights of FMU computations and communications depend on the FMU graph and the multi-simulation. We have run some size up experiments on our multi-simulation of heat transfer inside buildings. We have implemented larger simulations using greater number of computing nodes, replicating buildings on new nodes. Theoretically, the execution time of the multi-simulation should have remain almost constant (FMU computation time remained constant on each node, and communications were routed in parallel). Experimentally, Fig. 5 shows the execution time increase on PC clusters with 1 Gb/s and 10 Gb/s Ethernet interconnects. This time increase is more limited on the 10 Gb/s Ethernet interconnect. These experiments of heat transfer multi-simulations have pointed out the importance of the communications and the sensitivity to the interconnect speed.

#### 3.2.3 Difficulty to Fully Use Infiniband Interconnect

In order to reduce cost of these intensive and short communications, an interesting issue consists in using low latency and high bandwidth interconnects of standard HPC clusters, like some Infiniband networks. However, it requires to use some constrained middleware or communication libraries from HPC technologies (like MPI library), with native Infiniband interface. Using modern and confortable middleware (like ZMQ in DACCOSIM environ-

[5]FMUs were designed at EDF Lab Les Renardières using BuildSysPro models, and generated with Dymola 2016

[6]Sundials suite of nonlinear and differential/algebraic equation solvers, of the LLNL's Center for Applied Scientific Computing

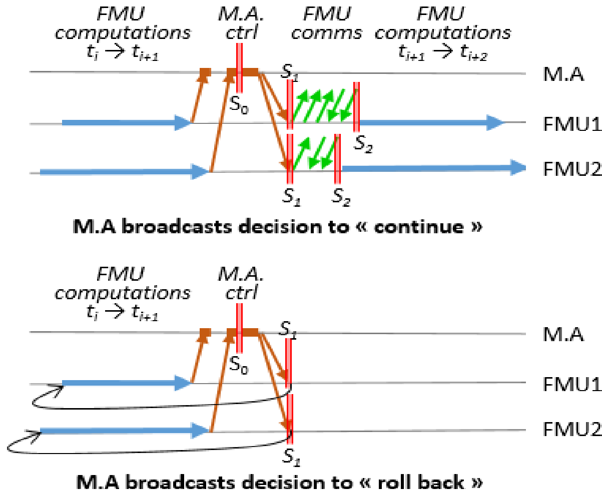


**Figure 6.** Relaxed synchronization of time step subparts

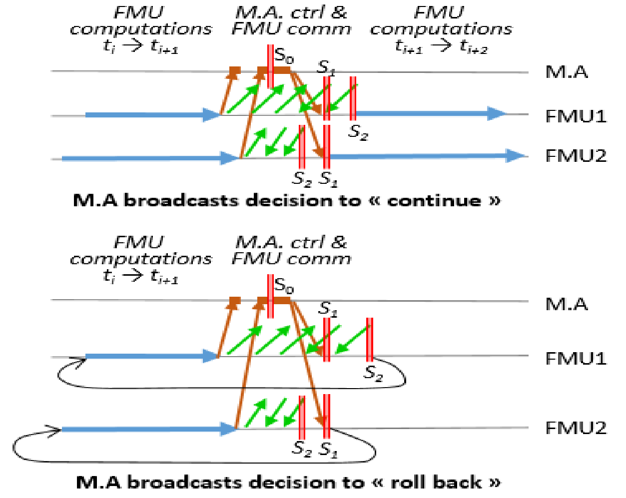


**Figure 7.** Overlapped orchestration mode

ment) leads to use Infiniband networks over TCP/IP adaptors and to loose their very low latency (Secco et al., 2014).

We attempted to use the MPI library to implement our multi-simulation communications, but MPI has been designed for process-to-process communications and appeared not adapted to DACCOSIM thread-to-thread communications, where each thread manages an FMU. We have currently suspended this investigation, and we use Infiniband networks over TCP/IP adaptors.

#### 3.2.4 Minimizing Message Sizes

Current communication mechanisms of DACCOSIM send FMU output data as strings, and send input name strings instead of short input identifiers (like input indexes). Future versions of DACCOSIM will implement shorter data encoding in order to reduce message sizes and bandwidth consumption.

### 3.3 Time Step Subparts Orchestration

#### 3.3.1 Ordered Orchestration with Relaxed Synchronization

Basic orchestration of a time step is illustrated on Fig. 6, and follows a *relaxed synchronization* mechanism. All FMU computations are run in parallel to progress from $t_i$ up to $t_{i+1} = t_i + h$, and as soon as an FMU has finished its computation substep it sends its requirements to the Master Algorithm (M.A.): to roll back and rerun with a smaller time step (to increase accuracy), to continue with the same time step, or to continue with a greater time step. Then, the M.A. processes each received requirement but awaits all requirements (synchronization point $S_0$) before taking a global decision, and broadcasting its decision to all FMUs. All FMUs wait for the M.A. global decision, and as soon as an FMU receives the M.A. decision (sync. point $S_1$), it rolls back or continues its time step.

- If an FMU receives the command to continue (top of Fig. 6), it enters its communication substep, sending its output results to connected FMUs and waiting for the update of all its input values (sync. point $S_2$). Finally, when it has updated all its inputs, it enters its next computation substep.
- If an FMU receives the command to roll back (bottom of Fig. 6), it restores its previous state at $t_i$ and reruns its computation step, but progresses from $t_i$ up to $t'_{i+1} = t_i + h'$, with $h' < h$ the new time step broadcasted by the M.A.

So, the only global synchronization barrier is the M.A. decision broadcast, that all FMUs are waiting for. Others synchronization points are *relaxed ones*, that stop only one task (the M.A. or one FMU). Then, each task going over a relaxed synchronization point carries on with its work independently of others tasks. Relaxed synchronization allows to increase performance, avoiding time consuming synchronization barriers and avoiding to synchronize all FMUs on the current slowest ones (the ones with longest computations or communications at current time step). Algorithms with relaxed synchronization schemes are usually more complex to implement and to debug, but ZMQ middleware has allowed an easy and efficient implementation of these communication and synchronization mechanisms between threads across a PC cluster.

#### 3.3.2 Overlapping Strategy

To still reduce the communication cost, a solution consists in overlapping some of the communications with some FMU computations, and with the *Master Algorithm* decision pending. Fig. 7 illustrates these mechanisms. When an FMU has finished its computation substep, it sends its requirements to the M.A. and, not waiting for M.A. decision broadcast, enters its communication substep. So, FMUs update their input values while the M.A. collects their requirements and broadcasts its global decision.

But, depending on the pending time of the M.A decison and on the number of inter-FMU communications, each FMU can cross its synchronization points $S_1$ (M.A. decision broadcast) and $S_2$ (all input update received) in any order (see FMU1 and FMU2 examples on Fig. 7). So, when both $S_1$ and $S_2$ synchronization points have been

crossed, each FMU considers the M.A. decision:

- If the M.A. has broadcasted a command to continue, then each FMU enters its new computation substep (see top of Fig. 7), and has saved some execution time achieving its inter-FMU communications while the M.A. decision was pending.
- If the M.A. has broadcasted a command to rollback, then each FMU waits for the end of its communications, restores its state at the beginning of the time step, and reruns its computation from $t_i$ up to $t'_{i+1}$. In this case, the overlapping mechanism has a little bit increased the execution time, achieving unnecessary inter-FMU communications.

From a theoretical point of view, our overlapping mechanism reduces the execution time when there are few rollbacks, or when using constant time steps. But from a technical point of view, some threads will work to send and receive messages while some threads will achieve the end of long FMU computations (M.A. decision broadcast is no longer a synchronization barrier). The communication threads could disturb the ongoing computations and slow down the multi-simulation, especially when running more threads than available physical cores (see section 3.1). Nevertheless, our overlapped orchestration mode has appeared efficient on our multi-simulation of heat transfer inside three floor building, run on a 6-core node cluster with a 10 Gb/s Ethernet interconnect. Section 5 will show the performance achieved on our benchmark of power grid multi-simulation.

### 3.4 Event Handling Impact

To increase their genericity, it seems necessary for CPSs to handle more signal kinds especially *continuous & piecewise differentiable* signals, *piecewise continuous & differentiable* signals and *piecewise constant* signals, which are sources of *events*. The current FMI-CS 2.0 release (Blochwitz et al., 2011) can theoretically approach events thanks to for example a bisectional search using variable step size integration (Camus et al., 2016). But only events due to piecewise constant signal changes can be detected. And the solution involves bad performance as it is based on rollbacks and finally some inaccuracies appear due to the last non zero integration step size.

We proposed to add new primitives in the FMI-CS standard (Tavella et al., 2016) in order to integrate hybrid co-simulation in a pure FMI-CS environment. Our solution does not require any model adaptation and allows to couple physics model with continuous variability and controllers with discrete variability. Moreover, parallelism is not reduced by our approach, as all FMUs continue to run concurrently either when processing shorter time steps, or when executing rollbacks. So, event handling by the FMI-CS evolution we have proposed does not require to change our parallel and distribution strategy of FMU co-simulation graph.

From a computing performance point of view, this FMI-CS standard improvement leads to execute a maximum of one rollback per FMU each time an unpredictable event appears during a time step. In the end, we do not know in advance how much the execution time will decrease but we are sure to achieve higher accuracy while improving the computation performance.

## 4 FMU Placement Strategies

### 4.1 Not a Dependence Graph Problem

A DACCOSIM program running a total of $n_F$ FMUs is composed of $n_F$ FMU wrapper tasks, $n_F$ data receiver tasks, plus a local or global control task per computing node (implementing our hierarchical M.A., see section 2.2). Of course, a DACCOSIM program can be considered as a dependency task graph, and some strategies exist to distribute such a graph on a PC cluster (Sadayappan and Ercal, 1988; Kaci et al., 2016). However, a DACCOSIM task graph has some specific task dependencies. During one time step in ordered orchestration mode, all FMUs execute three substeps as illustrated on Fig. 8:

- The computation substep (Fig. 8 part *a*): all FMU wrapper tasks run concurrently and autonomously, achieving the FMU computations. There is no dependence between these tasks during this substep. The only optimizations consist in load balancing the FMU computations among the computing nodes, and to set only $n_c - 2$ FMU per nodes when there are enough available computing nodes, according to section 3.1.
- The control substep (Fig. 8 part *b*): each FMU wrapper task sends its wish to the control task for the next operation (rollback or continuation, and size of the future time step) and waits for its global decision. There is a total dependence of the control task to all the wrapper tasks, followed by a total dependence of all the wrapper tasks to the control task, close to a *synchronization barrier* for the FMU wrapper tasks (see section 3.3). There is no optimization to achieve when distributing the FMU graph, excepted to implement a local control task on each node to manage its FMU wrapper tasks.
- The communication substep (Fig. 8 part *c*): it is only achieved when no rollback is ordered by the control task. Each FMU wrapper task sends its new outputs to connected FMU inputs, while each FMU receiving task ensures the reception of the new input values of the FMU. These communication operations are not CPU consuming. So, we run in parallel up to $2 \times n_f$ tasks on each computing node hosting $n_f$ FMUs, in order to optimize the communications (see section 3.3). All these tasks run without any synchronization nor dependence during the communication substep, and the only optimization when distributing the FMU graph consists in grouping on the same node the most strongly connected FMUs (fighting with the load balancing objective).

In fact, we can classify our DACCOSIM task graph as a *periodic task graph*. Its period is equal to one time step,

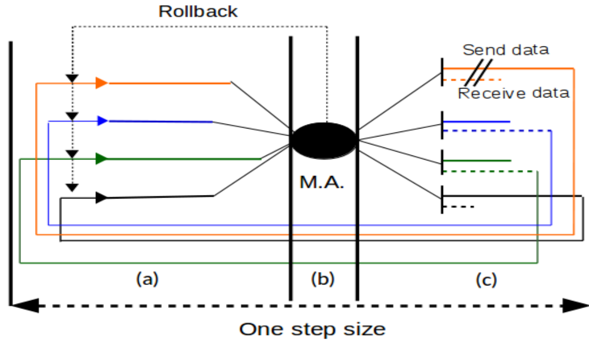


**Figure 8.** Multi-task synchronization overview (*ordered* mode)

and includes 2 phases (*a* and *c*) with pure concurrent task executions, and one phase (*b*) which is a kind of synchronization barrier (with only the control task working). So, we do not consider the task dependencies to distribute our FMU graph on a PC cluster. We focus on load balancing, on grouping the most connected FMUs, and on limiting the number of FMU per nodes (when there are enough available nodes).

IFP EN and INRIA succeeded to parallelize computation inside wide FMUs thanks to a fine scheduling of basic operation executions on one multi-core node (Saidi et al., 2016). The practical speed-up observed by our colleagues is achieved by imposing a constrained allocation of all the operations of a same FMU to the same core. Their approach is complementary to ours as they optimize the co-simulation of FMUs on the different processors of the same calculation node while we are optimizing the deployment of a calculation graph composed with lots FMUs on a possibly wide set of multi-core nodes.

## 4.2 Different Contexts and Approaches

The main trouble to establish a good distribution of the FMU graph on a PC cluster is the lack of metadata about FMU computations in the FMI-CS standard. There is no information about FMU computation time, or computation complexity. Dynamic load balancing is out of reach of our current implementation, and static load balancing of the computations on the nodes of a PC cluster remains difficult. This section introduces the different approaches we identified to distribute FMU graphs.

### 4.2.1 Previous Experimental Approach

In the beginning of 2016 we distributed on two PC clusters a first multi-simulations of heat transfers inside buildings. Each building was a subgraph of only 10 FMUs. We ran a small one-building problem setting only one FMU per node, so that FMUs could run the real simulation without disturbing each other (not sharing cache memory, nor memory bandwidth, nor cores. . . ). We measured the computation time of each FMU (to *characterize* our different FMUs), and then we established the most load balanced FMU distributions on various number of nodes. Finally, some complementary experiments allowed to identify the most efficient distribution of a one-building problem on each PC cluster.

When the best distribution of a one-building problem (using $n_0$ nodes) was identified, we enlarged the problem with $k$ buildings, replicating our best distribution (using $k \times n_0$ nodes). We successfully *scaled up* (Dad et al., 2016): processing larger problem on larger cluster in similar time. But this approach takes too long development times, and replicating the best elementary distribution leads to use too many nodes, some cores remaining unused. This approach cannot be a generic solution.

### 4.2.2 Approach function of the User Knowledge

From our point of view, distributing a totally unknown FMU graph or a fully characterized FMU graph should be infrequent DACCOSIM use cases. Users build co-simulations based on their expertise and have an initial knowledge about computation loads and communication volumes in their FMU graphs, allowing to use basic distribution mechanisms. When testing and improving their co-simulations they accumulate knowledge on their FMU graphs, and can use more complex heuristics. However, it is not obvious to design an efficient heuristic.

During the development phase of a co-simulation many FMU graphs are only run a few times and the FMU graph distribution has to be computed quickly, without long calibration steps. But when a co-simulation design is finished and successful, it can enter a long exploitation phase, requiring frequent runs. Then, it can be profitable to make detailed performance measurements and to compute a fine distribution of the FMU graph, in order to use less computing nodes and/or to decrease the co-simulation time.

Considering current and future usage of DACCOSIM at EDF, for smart grid co-simulations, we identified 3 levels of user knowledge, and we propose 3 associated generic FMU distribution approaches.

**a - Identifying FMU Families**: when users have only minimal technical and skill information about their co-simulations, and are able just to group the FMUs in families with close computing load.

*Proposed approach*: each family will form an FMU list, and the concatenated list of all FMU families will be distributed on the computing nodes according to a round-robin algorithm. This approach requires light knowledge on the FMUs used, and succeeded to load balance the FMU computations on our benchmark (see section 5).

*Extreme use case*: If no information is available on some external FMUs, it is possible to group these FMUs in a particular family to spread over the computing nodes. If no pertinent information is available on any FMUs, it is also possible to group all FMUs in a unique family, to achieve a random distribution and to track a statistical load balancing.

**b - Cumulating Knowledge for Heuristics**: when users progress in their co-simulation development they improve their knowledge about their FMUs and FMU graph. This extra-knowledge can be exploited by more or less generic heuristics to improve the FMU distribution. For example:

- running and testing different configurations of the FMU graph allows to learn some relative computing weights (ex: $t_{FMU2}^{comput} \approx 0.5 \times t_{FMU1}^{comput}$),
- analyzing the FMU graph allows to detect some regular patterns strongly connected (ex: a city area connected on one medium voltage network of a smart grid).

*Proposed approach*: design and use an heuristic (1) to optimize load balancing in order to reduce the global computation time, and/or (2) to group on same nodes the FMUs strongly interconnected in order to reduce the global communication time.

*Warning*: our experiments have shown the load balancing optimization is the most important criterion, however designing an efficient heuristic (improving performance of the previous approach) remains difficult.

**c - Building Models of Computation and Communication Times**: when an FMU graph enters an exploitation phase, it can be profitable to establish an execution time model of the co-simulation, to optimize its distribution and the computing resource usage.

*Proposed approach*: (1) run smaller but similar co-simulations, deploying only one or very few FMUs per node (to avoid mutual disruption between FMUs) and measure each FMU computation time on the nodes of the target cluster, (2) analyse the FMU graph to compute the volume of each inter-FMU communication, and measure the experimental applicative latency and bandwidth on the target cluster. Then, establish a computation and a communication time model of the co-simulation, to feed a solver looking for the best distribution of the FMU graph.

*Warning*: This approach requires long experimental measurements.

The IDEAS test case described in section 5, has been distributed on different PC clusters according to the *Identifying FMU families* and the *Cumulating knowledge for heuristics* approaches.

### 4.3 FMU Distribution on Virtual Nodes

When the FMU graph is defined, the DACCOSIM software suite distributes the FMUs and generates Java source files for a set of *virtual nodes*, and maps the *virtual nodes* on the available *physical computing nodes* at runtime. Then the Java source codes are compiled, a JVM is started for each virtual node and its Java program is executed. We defined intermediate *virtual nodes* in order to generate Java source files independent of physical node names and IP addresses, and to make the deployment more flexible on nodes with Non Uniform Memory Architecture (NUMA).

Modern computing nodes have several processors and memory banks interconnected across a small network. But memory access time becomes function of the distance between the core running the code and the memory bank storing the data (NUMA principle). Creating one process (one JVM, one virtual node) per NUMA subnode instead of creating a unique JVM per node managing all the threads, can increase data locality and performance.

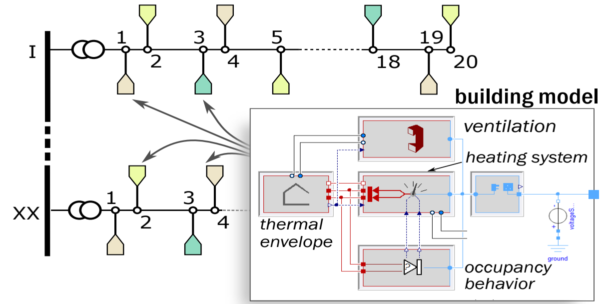


**Figure 9.** Topology of the large scale testbed using IDEAS lib.

## 5 Large Scale Experiments

### 5.1 Experiment Objectives

The co-simulation of a large scale District Energy System was chosen as a testbed. Co-simulation methods are foreseen to handle several bottlenecks encountered during CPS simulation on one single simulation tool, such as:

- Multi-Physics integration (electrical, hydraulic, thermal, etc.),
- Multiple time-scales and dynamics,
- Implementation of controllers,
- Scalability, i.e. the capability to study a growing number of buildings and the growing size of the power grid.

The numerical experiment consists in a complex multi-physical district energy system. The main purpose of this section is thus to propose a proof of concept of co-simulation with lots of FMUs on a HPC cluster and to highlight the advantages of such an approach for large scale systems.

### 5.2 Testbed Description

In this section, we propose to assess DACCOSIM Master Algorithm efficiency by co-simulating an electrical distribution grid using a variable number of cluster nodes. The model has been completely implemented using Modelica and the OpenIDEAS library[7] (Baetens et al., 2015). Neither the electrical grid, the heating systems nor the building envelops have been simplified.

The general structure of the use case is shown on Fig. 9. It is composed of 1000 buildings connected to low voltage (LV) feeders, each of them including a thermal envelope, ventilation and heating systems and a stochastic occupancy behavior. The buildings are dispatched on 20 low voltage LV feeders, each modeled as one FMU, noted I to XX on Fig. 9. These feeders are connected to a medium voltage (MV) network that is also simulated with a single FMU. A data-reading FMU provides real medium voltage measurements that are imposed at the MV substation busbar. The electric grid frequency is provided to different FMUs (buildings and feeders) by 20 additional FMUs.

[7]EFRO-SALK project, with support of the European Union, the European Regional Development Fund, Flanders Innovation & Entrepreneurship and the Province of Limburg

This distributed frequency FMU implementation is meant to reduce inter-node communications since the frequency has to be dispatched to all the FMUs of the use case. Finally, the co-simulation holds a total of 1042 FMUs exported from Dymola 2016 FD01 in conformance with the FMI-CS 2.0 standard. A smaller use-case with less buildings and only 442 FMUs, has also been designed to evaluate the scalability of our solution.

The test case was run on two different clusters: (1) *Sarah* at CentraleSupelec Metz, composed of dual 4-cores Intel Xeon E5-2637 v3 at 3.5 GHz (Haswell) with a 10 Gb/s Ethernet communication network, and (2) *Porthos* at EDF R&D, composed of dual 14-cores Intel Xeon E5-2697 v3 at 2.60 GHz (Haswell) with Infiniband FDR communication network. These clusters are labeled "sar" and "por" on performance curves of section 5.4. On both clusters, DacRun is used to deploy and run the DACCOSIM co-simulation. DacRun is implemented in Python 2.7, is compliant with OAR and SLURM cluster management environments, and can also be used on standalone machines (for small experiments). It achieves Java source files compilation, virtual/physical nodes mapping, JVMs starting and can ensure to gather the results and logs.

## 5.3 Numerical Results

The runs are done for a one-day simulation with one-minute constant step size. The co-simulation gives realistic results according to expert judgment. Moreover the energy consumption of the buildings follows the same trend as the one observed on a Dymola simulation limited to one 20-building feeder. To assess the correctness of the co-simulation on cluster, we selected a single building of the test case and simulated it with Dymola by injecting sampled voltage data obtained from the cluster co-simulation. The power consumed by the building simulated with Dymola and the one co-simulated on cluster should be the same as the two selected buildings have the same environment: same input voltage, same weather data and same occupancy data. The root mean square error on the current between those two simulations is $1.16 \times 10^{-2}$ A, with current mainly in the range $1 - 10A$. The two currents for the one-day simulation are plotted on top of Fig. 10 with a close-up on its bottom. The dynamic of the power consumption is well reproduced thus the cluster co-simulation seems reliable.

## 5.4 Performance and Scaling

The FMUs were dispatched on the nodes following two different approaches introduced in section 4.2: with (1) a *Cumulated knowledge for heuristic* approach exploiting the problem topology with balanced load ("KHBL" on Fig. 11), and (2) according to an *Identifying FMU families* approach associated to a round-robin mechanism ("FFRR" on Fig. 11). Experimentations were conducted on our clusters in the ranges $32 - 1024$ and $112 - 1792$ cores, with *overlapped* and *ordered* orchestration modes ("over" and "order"), on both 442 and 1042 FMUs use-cases.

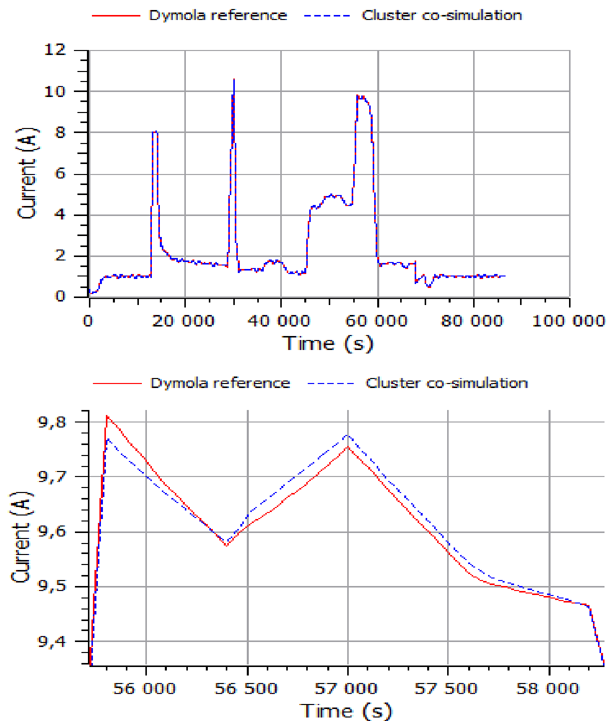


**Figure 10.** Current from a building of the DACCOSIM co-simulation and its Dymola counterpart

**Scalability Achievement:** time curves on Fig. 11 appear very linear on this full logarithmic scale graphic, slope is close to $-1$ on HPC Porthos cluster, and time curves of different problem sizes are parallel. So, execution time regularly decreases when using more cores, and similar performance can be achieved when running larger problems on larger number of cores (from 442 to 1042 FMU benchmark curves). Of course, when using as many cores as FMUs the execution stops to decrease (right-hand side of Porthos curves).

**Interconnect and Communication Impact:** time curve slope is smaller on Sarah cluster and its 10 G/s Ethernet interconnect, than on Porthos and its high performance Infiniband FDR interconnect. Communications are not negligible, and a high performance interconnect (low latency and high bandwidth) improves the scalability.

**Complex Choice of the Orchestration Mode:** Overlapped mode was the fastest one on a previous use case run on a cluster with smaller nodes (Dad et al., 2016). But when running IDEAS use case on Sarah cluster, the overlapped mode appears slower than the ordered one, and when run on Porthos cluster, both orchestration modes have close performances up to allocate enough nodes to get one core per FMU. Beyond this limit it remains free cores on each node to manage communications, and the execution time of the overlapped mode roughly decreases and really becomes the smaller one. So, both orchestration modes are interesting, but strategy to foresee the right one is still under investigation.

**Difficulty to Design Efficient Heuristics:** our heuristic based on FMU graph knowledge, aiming to group connected FMUs on the same node with respect to load bal-

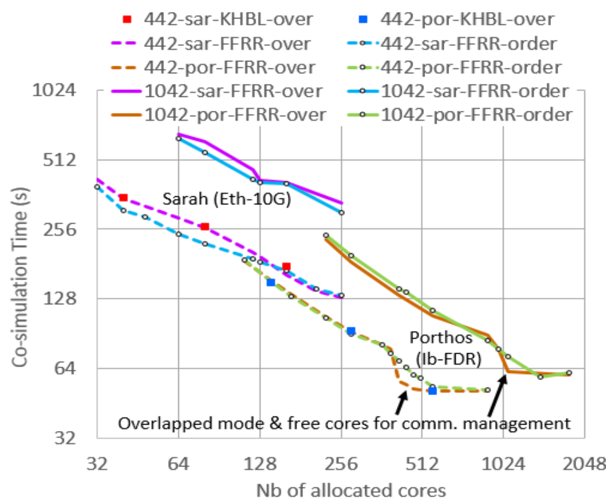


**Figure 11.** IDEAS benchmark with 442 FMUs run on clusters

ancing, requires in our testbed some accurate number of nodes (5, 10 or 20 on our example) and does not achieve better performance than round-robin distribution of FMU families. An efficient heuristic remains hard to design and our round-robin on FMU families algorithm appears a good solution

## 6 Conclusion and Perspectives

With DACCOSIM generating Java files for Linux and its Python add-in DacRun easily compiling, running and collecting the results of a DACCOSIM application on clusters, we have illustrated in this paper the capability of our FMI-CS based environment to manage very wide co-simulations. Our testbed is a realistic case study using the OpenIDEAS library and involving the detailed modeling of 1000 buildings scattered on a distribution grid. We have demonstrated the feasibility of scaling-up the multi-simulation by pushing very far the limits of the simulation and taking advantage of Porthos, the EDF cluster ranked 310th in the 48th edition of the TOP500 list published in November 2016.

Work is currently being carried out to further improve the capabilities of our co-simulation tools suite. Some can be performed with the current FMI-CS 2.0 standard (e.g. minimizing inter-FMU message sizes), while others would require an evolution of the standard (e.g. event handling of accurate hybrid co-simulation).

A collection of generic heuristics for FMU graph distribution, when knowledge on co-simulation has been accumulated, is also under development, to make easier large scale deployments of more complex co-simulations.

## References

R. Baetens, R. De Coninck, F. Jorissen, D. Picard, L. Helsen, and D. Saelens. OPENIDEAS - An Open Framework for Integrated District Energy Simulations. In *Proceedings of Building Simulation Conference 2015 (BS 2015)*, Hyderabad, India, December 2015.

T. Blochwitz, M. Otter, M. Arnold, C. Bausch, C. Clauß, H. Elmqvist, A. Junghanns, J. Mauss, M. Monteiro, T. Neidhold, D. Neumerkel, H. Olsson, J.-V. Peetz, and S. Wolf. The Functional Mockup Interface for Tool independent Exchange of Simulation Models. In *Proceedings of the 8th International Modelica Conference*, Dresden, Germany, March 2011.

B. Camus, V. Galtier, and M. Caujolle. Hybrid Co-Simulation of FMUs using DEV and DESS in MECSYCO. In *Proceedings of the 2016 Spring Simulation Multiconference, Symposium on Theory of Modeling and Simulation (TMS/DEVS'16)*, Pasadena, CA, USA, April 2016.

K. M. Chandy and J. Misra. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Trans. Softw. Eng.*, 5(5), September 1979.

C. Dad, S. Vialle, M. Caujolle, J.-Ph. Tavella, and M. Ianotto. Scaling of Distributed Multi-Simulations on Multi-Core Clusters. In *Proceedings of 25th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2016)*, Paris, France, June 2016.

V. Galtier, S. Vialle, C. Dad, J.-Ph. Tavella, J.-Ph. Lam-Yee-Mui, and G. Plessis. FMI-Based Distributed Multi-Simulation with DACCOSIM. In *Proceedings of the 2015 Spring Simulation Multiconference, Symposium on Theory of Modeling and Simulation (TMS/DEVS'15)*, USA, April 2015.

A. Kaci, H. N. Nguyen, A. Nakib, and P. Siarry. Hybrid Heuristics for Mapping Task Problem on Large Scale Heterogeneous Platforms. In *Proceedings of the 6th IEEE Workshop on Parallel Computing and Optimization (PCO 2016), IPDPS Workshop 2016*, Chicago, IL, USA, May 2016.

A. Ricci, M. Viroli, and A. Omicini. Give Agents Their Artifacts: The A&A Approach for Engineering Working Environments in MAS. In *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'07)*, Honolulu, HI, USA, May 2007. ACM.

P. Sadayappan and F. Ercal. Cluster-partitioning Approaches to Mapping Parallel Programs Onto a Hypercube. In *Proceedings of the 1st International Conference on Supercomputing (ICS 1988)*, Athens, Greece, June 1988. Springer-Verlag.

S. E. Saidi, N. Pernet, Y. Sorel, and A. Ben Khaled. Acceleration of FMU Co-Simulation On Multi-core Architectures. In *Proceedings of 1st Japanese Modelica Conference*, Tokyo, Japan, May 2016.

A. Secco, I. Uddin, G. P. Pezzi, and M. Torquati. Message Passing on InfiniBand RDMA for Parallel Run-Time Supports. In *Proceedings of the 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP 2014)*, Turin, Italy, February 2014.

J.-Ph. Tavella, M. Caujolle, S. Vialle, C. Dad, Ch. Tan, G. Plessis, M. Schumann, A. Cuccuru, and S. Revol. Toward an Accurate and Fast Hybrid Multi-Simulation with the FMI-CS Standard. In *Proceedings of the IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA 2016)*, Berlin, Germany, September 2016.

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