A Conservative Approach to SystemC Parallelization

B. Chopard[1], P. Combes[1], and J. Zory[2]

[1] University of Geneva - Department of Computer Science, Switzerland
[2] STMicroelectronics - AST, Geneva, Switzerland

Abstract. SystemC has become a very popular language for the modeling of System-On-Chip (SoC) devices. However, due to the ever increasing complexity of SoC designs, the ever longer simulation times affect SoC exploration potential and time-to-market. We investigate the use of parallel computing to exploit the inherent concurrent execution of the hardware components, and thus to speed up the simulation of complex SoC's. A parallel SystemC prototype based on the open source OSCI kernel is introduced and preliminary results are discussed.

1 Introduction

The design of modern Systems-on-Chips (SoC) becomes more and more demanding as the complexity and the functionality of new circuits and applications increase. The ability to develop and test these systems with completely virtual platforms and reasonably fast simulations is a key enabler for tomorrow's technology. The SystemC language was developed by the Open SystemC Initiative (OSCI) to enable system level design. It has quickly become a very popular modeling solution, for engineers can represent the functionality, communications, software and hardware components at multiple levels of abstraction with a single common language.

This paper explores the use of parallel techniques to speed up the simulation of SoC executable specifications. Section 2 describes a parallel SystemC kernel prototype built from the public domain OSCI simulator. A performance analysis is then derived from a straightforward pipeline application in Section 3. Finally, experimental results of the parallel simulation of a complex telecom application are presented in Section 4.

2 A Parallel SystemC Kernel

SystemC [1] exhibits two features that motivated our work. First of all, SystemC is most often used to describe the behavior of a complete system where several hardware and/or software components concurrently perform some tasks. The purpose of this work is to exploit this inherent concurrency to implement parallel simulations.

The second important feature of SystemC is its C++ open source library. The SystemC paradigm combines the flexibility, portability and ease-of-use of object-oriented C++ programming with dedicated concepts and constructs for SoC modeling practices [2]. Our choices in developing a parallel SystemC framework are driven by the willingness to keep the modeling style as open as possible.

2.1 The OSCI SystemC Kernel

A typical SystemC application is characterized by both a structural part and a behavioral part. The structure will basically describe the way (hardware and software) components are connected to each other ; this translates into various modules connected via channels. Hierarchical structures are supported as well: modules may instantiate other (sub-)modules. The behavioral part of the system is captured in SystemC with the notion of process. Each module may contain one or several processes. The execution of a given process is driven by events (such as a value change on the channels connected to the module) that might awaken or even restart the process.

Once the user has fully described both aspects of the application, it is the task of the SystemC kernel to simulate the whole system, given certain input stimuli. The kernel has to schedule, on a sequential processor, the multiple processes of the system in response to those events generated by the application.

To make sure the concurrent processes use coherent inputs/outputs, the scheduler runs all the processes that are ready to be executed first, and only then it updates their changes on the channels. Both evaluate and update phases constitute a δ-cycle: multiple δ-cycles can occur at the same simulation time, which is very useful for modeling fully-distributed, time-synchronized computation (as in Rgeister Transfer Level) [1].

In the OSCI open source kernel, the scheduler relies on a stack of “runnable processes” (cf. Figure 1) that is emptied during the evaluate phase, and then filled again in with those processes that are sensitive to any event triggered during the update phase (such as a “value-changed” event on a channel). If the stack is empty at the end of the update phase, then the simulation time can be updated to the time of the next event to be triggered, and if there are none, the simulation stops.

2.2 Towards a Parallel SystemC Kernel

There exists some distributed SystemC platforms ([3, 4]) which aim at coordinating various simulators over geographically distant sites. They allow Intellectual Property (IP) vendors to expose their products for testing by potential clients, while hiding their behavior details. Hence, the primary goal is not performance, but usability, whereas we look for fast simulations at the scale of cheap clusters of PC’s.

In [5], Savoui et al. encapsulate SystemC processes into POSIX threads to transparently benefit from shared memory SMP architectures. This is integrated in the last version of the OSCI kernel, but complex designs would require costly huge computers.

To our knowledge no true parallel implementation of SystemC exists so far. However much work has been done in the last two decades about parallel VHDL or Verilog simulation kernels, two famous Hardware Description Languages. It is related to the wider field of Parallel Discrete Event Simulations (PDES). Indeed, VHDL or Verilog applications, as well as SystemC applications, are conceptually Discrete Event Systems (DES) [6, 7]: in short, upon the occurrence of some events, processes are awaken and some computation produce in turn new events. There is a large body of literature about PDES, their implementation [8, 9, 10] and their utility for parallel Verilog or VHDL simulation [11, 12, 13], but they have not yet been introduced for SystemC.

The so-called conservative approach strongly enforces causality constraints and thus requires a lot of synchronizations between the interacting processes located on different computing nodes. The so-called optimistic approach is more permissive as it allows processes not to synchronize as often as they strictly should, using forecast mechanisms. When a causality error is detected, a rollback procedure is used.

Optimistic PDES are rather complex and require knowledge of the underlying concepts for the tuning of many application dependent parameters [14]. This constraint is a major drawback for SoC designers, and thus these PDES are not widespread in their community [15]. Furthermore, as VHDL/Verilog applications are mostly low-level abstraction designs, rollbacks can be cost-effective, for little work is to be undone, but with large data sets and coarse grain processes such as in functional-level SystemC models, these mechanisms would consume tremendous amounts of memory, whereas the reduced synchronization/computation ratio makes the overhead for conservative synchronizations acceptable. For these reasons we consider here a conservative approach when parallelizing SystemC. If the CPU time between successive synchronizations is long enough and a fair load balancing among the processors can be achieved, we may expect a reasonable speedup from this approach.

In order to have an open source kernel, to preserve compatibility with SystemC libraries and semantics as well as to keep SystemC users still feel familiar with a parallel kernel, we choose to implement the conservative algorithms directly in the OSCI open source reference kernel.

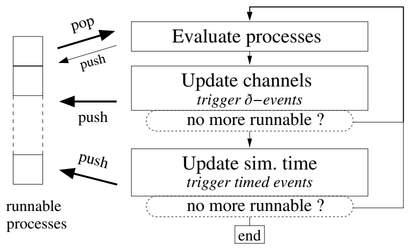


Fig. 1. SystemC kernel scheduler

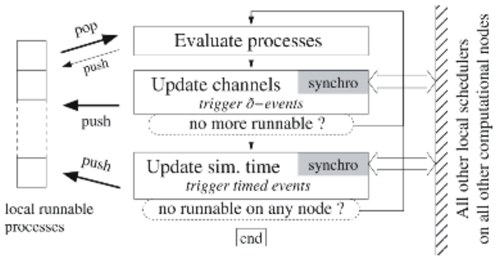


Fig. 2. Scheduler local to one node

2.3 Parallelization of the OSCI Kernel

Given the goals discussed in section 2.2, our parallel kernel has to face two major issues: respect for the informal SystemC semantics and minimum changes in the API (ie. the language itself), while still offering the flexibility to specify the process partitioning.

Our approach is to have a copy of the OSCI SystemC scheduler running on every processing node, each simulating a subset of the application modules. The SystemC semantics specify that, within a δ-cycle, the execution order of the processes is not pre-defined[1] [1], but which processes are to be run depends on the previous δ-cycle. Thus, within a δ-cycle, it is possible to execute the processes in parallel, but all local schedulers must synchronize at its end. The conservative approach discussed in section 2.2 leads us to implement a strong synchronization of both channels and time (Figure 2).

[1] But it is the same for two executions of the same simulator, so that some dependency bugs can be hidden; this is still true on every node of the parallel simulator, but not at the global scale.

The synchronization of one channel only involves the nodes that it connects: if its value has changed, the nodes trigger the associated event so that the sensitive processes are pushed back in the local runnable processes stacks, like in a serial kernel.

The second synchronization deals with simulation time. Our implementation defines a master node which collects the next timed events from every other node, computes the next simulation time and then sends it to all nodes for local update. If this time is the current simulation time, then it means that the current δ-cycle loop has not finished yet on some node: all other nodes must run another δ-cycle, even if no process is to be run locally. The end of the simulation is reached (.i.e. the local schedulers can stop) when the simulation time has reached its maximum value.

With regard to the changes in the language, they are limited to the addition of a new kind of sc_module: sc_node_module. From the user point of view, a "node module" looks like a classical purely hierarchical module, except that it cannot have other node modules as submodules. Internally, it gathers all the modules that are to be managed by one processing node; the partitioning of the application is thus the responsibility of the designer. The channels connecting node modules are internally duplicated, and their bindings to the node modules ports fully define how to synchronize them. It has been a major reverse-engineering and development work to build, from this very simple way of partitioning, all the information necessary to perform the required synchronizations, but this would be too technical to be reported here in details.

As a conclusion, a functional parallel kernel has been developed and validated against a subset of the OSCI test suite ; few very specific SystemC features are not yet fully supported but this is no limitation for most applications.

3 Performance Analysis with a Basic Pipeline

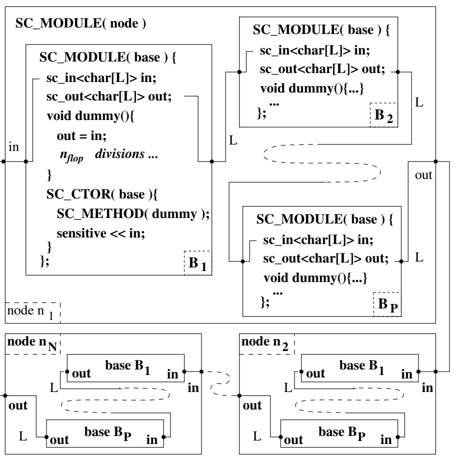
In this section, we validate the concept of our parallel SystemC kernel and its prototype implementation against a regular pipeline test-application.

3.1 Description of the Application

Our test-application is a pipeline loop made of $N \times P$ stages distributed in N modules and P submodules. In the parallel simulator, N is also the number of computational nodes. Every (sub)module has one input and one output port, interconnected as shown Figure 3. The channels used are sc_signal⟨char[L]⟩'s: they hold two versions of an L-long array, the current one and the one for the next update - see 2.1. Every submodule on its turn defines a process which is sensitive to its input signal; each time the process is awaken (actually, every δ-cycle), the behavior is to "increment" and copy data from input to output and to perform n_{flop} floating point divisions to emulate CPU load.

3.2 Performance Model

The model below uses the terminology defined in Figure 3. It only considers CPU processing time and communications over the network. δ-cycle synchronizations and possible overheads introduced by the sequential and parallel SystemC kernels are on-purpose not addressed in this "reference" model. Potential discrepancies between this



```
SC_MODULE( node )
  SC_MODULE( base ) {
    sc_in<char[L]> in;
    sc_out<char[L]> out;
    void dummy(){
      out = in;
      n_flop divisions ...
    }
    SC_CTOR( base ){
      SC_METHOD( dummy );
      sensitive << in;
    }
  };
```

```
SC_MODULE( base ) {
  sc_in<char[L]> in;
  sc_out<char[L]> out;
  void dummy(){...}
  ...
};
```

N : Number of modules / computing nodes
P : Number of submodules
L : Size of data transferred on the signals at every δ-cycle
n_{flop} : Number of dummy floating point divisions per process and δ-cycle
n_δ : Number of δ-cycles executed for the simulation
α : CPU time to execute one floating point division
β : Transmission time for one byte over the network
γ : Time to access one byte in memory
t_{st} : Time for initializing a communication

Fig. 3. Structure and parameters of the application

model and the simulation results would demonstrate the impact of SystemC scheduling in the overall performance.

The total execution time can be written as : $T = t_{init} + n_\delta . t_\delta + t_{finish}$. If n_δ is large enough, then t_{init} and t_{finish} can be neglected. Thus they are ignored by our timers for the experiments. Assuming that the scheduler overhead is null, one has

$$t_{\delta_{seq}} = N(t_{evaluate} + t_{update_{seq}}) \qquad t_{\delta_{//}} = t_{evaluate} + t_{update_{//}}$$

Here $t_{evaluate}$ represents the computation time for the n_{flop} divisions in all P submodules, added to the time of the memory accesses for the data copies and increments:

$$t_{evaluate} = P(\frac{n_{flop}}{\alpha} + \frac{2L}{\gamma} + \frac{2L}{\gamma})$$

The intra-signals updates consist of $(P-1)$ copies. The inter-signals updates still consist of copies, but in the parallel version, there are also network communications:

$$t_{update_{seq}} = (P-1)\frac{2L}{\gamma} + \frac{2L}{\gamma} \qquad t_{update_{//}} = (P-1)\frac{2L}{\gamma} + \frac{2L}{\gamma} + (t_{st} + \frac{L}{\beta})$$

When all terms are gathered, the performance model is:

$$T_{//} = n_\delta \left[\frac{6}{\gamma} LP + \frac{L}{\beta} + \frac{n_{flop}}{\alpha} P + t_{st} \right] \quad (1) \qquad T_{seq} = n_\delta NP \left(\frac{6}{\gamma} L + \frac{n_{flop}}{\alpha} \right) \quad (2)$$

3.3 Model Validation

All our experiments run on a cluster of up to 52 1.5GHz Pentium IV mono-processor nodes, with 500MB RAM, and connected through a Fast-Ethernet network.

A pure MPI/C++ version of the pipeline test-application was first developed to serve as a reference. This version simply avoids all the possible SystemC scheduler overheads and hence almost perfectly matches the performance model described above. For space reasons and to put a stress on the SystemC version, no figure is given here to illustrate this, but our experiments clearly highlight that $T_{seq}(P)$ and $T_{seq}(N)$ are linear, and that $T_{//}(P)$, $T_{seq}(L)$ and $T_{//}(L)$ are affine. Furthermore, the slope and intersect values of the lines approximate quite well the expected cluster hardware performance, according to the model. Yet, network congestions (low bisectional bandwidth) occur when more and more communications are requested synchronously, i.e. when N grows, and thus an additional dependency upon N may appear on the intersect of $T_{//}(P)$.

Globally, the SystemC version of the pipeline gives similar results. This is particularly true for T_{seq} that still matches the model very well (data not shown here). According to Figure 4, $T_{//}(L)$ is obviously affine too, and the slopes and the intersects of the lines figure out the influence of P and n_{flop}. However, they also reveal a slight dependency on N. It cannot be explained by network congestion only, as for the MPI version, because the impact is more visible : in the SystemC version, after every δ-cycle, an additional synchronization is performed, to manage the end of the δ-cycle loop (see section 2.3). Although it was ignored in the current performance model, it is expected to grow at least as $\log N$ as it requires an exchange of information with a master node.

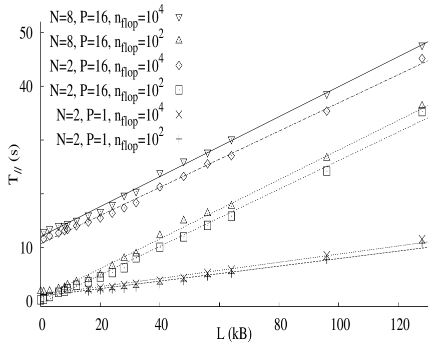


Fig. 4. $T_{//} = n_\delta \left[\left(\frac{6P}{\gamma} + \frac{1}{\beta} \right) L + P \frac{n_{flop}}{\alpha} + t_{st} \right]$

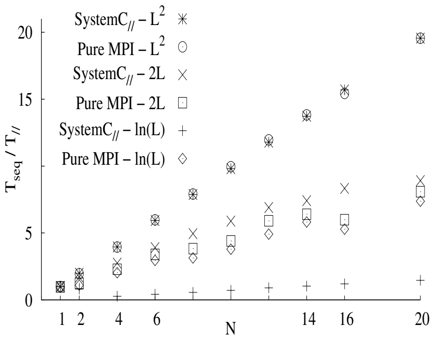


Fig. 5. Speedups (with L=1000)

3.4 Performance

Figure 5 compares the speedups of the reference (namely “pure MPI”) and the SystemC versions, with realistic values for n_{flop}: indeed, the amount of computation is often dependent on the amount of data. We have tested three kinds of such dependencies, following the complexities of usual algorithms: $\ln(L)$, $2L$ and L^2.

To explain the shape of the curves, let us consider the efficiency $E = T_{seq}/T_{//}$. Equations 1 and 2 of the performance model lead to the relation :

$$P \left(\frac{6\beta}{\gamma} + \frac{\beta}{\alpha} \frac{n_{flop}}{L} \right) = \frac{E}{1-E} \left(\frac{1}{\beta} + \frac{t_{st}\beta}{L} \right)$$

With our cluster, $\frac{6\beta}{\gamma} \approx \frac{\beta}{\alpha}$, and since our tests run with $L = 1000 \gg 1$, one has $\frac{t_{st}\beta}{L} \ll 1$. Hence, when $n_{flop} = O(\log(L))$ or $O(L)$, $E \approx \frac{1}{1+\frac{1}{P}} < 1$: the curve is “attracted” by the

N axis, for P cannot grow too much. Yet, when $n_{flop} = O(L^2)$, $E \approx \frac{1}{1+\frac{1}{LP}} \approx 1$. This not only explains the general shape of the curves but also why the $\ln(L)$ and $2L$ results are so close for the pure MPI version. For the parallel SystemC version however, the "time synchronizations" (see section 2.3), ignored in our model, become significant if n_{flop} is very small ($\ln(1000) \approx 7$).

4 Case Study: Beyond 3G Modem

The previous section demonstrates that the achievable speedup of a distributed SystemC application heavily depends on the number of processors and the computation to communication time ratio. Here, we further explore those aspects with a real complex application, developed in the frame of the IST MATRICE project. This project involved many academic and industrial partners, from the field of cellular telecommunication Research and Development. It aimed at investigating "Multi Carrier-Code Division Multiple Access" techniques for the broadband component of Beyond 3G systems.

The platform is based on a pipeline, composed of three major stages, made of multiple SystemC modules, which model the major physical layer components of a wireless transmitter, receiver and channel. Please see [16] for a full description of this application. It is only worth noting here that it implements more than 26000 lines of C++ style code, with 27 channels and 26 processes.

A first parallel version of this application is almost immediately available. Indeed, the canonical partition is to have every of the three major stages running on its own computing node. Thus, the work is limited to replacing the SC_MODULE keyword by SC_NODE_MODULE (see section 2.3) in the declarations of these three modules. However, as it could be already anticipated from the pipeline performance model, a good speedup can only be achieved when CPU loads are well balanced over the computational nodes. Some quick investigations revealed that one of the three stages already accounts for 47% of the total serial simulation time. Thus, with the 3-node canonical partition, the speedups are intrinsically limited to 2.1. We reached a speedup of 1.92.

To overcome this heterogeneity, a 4-node partition has been implemented, with an effort of gathering the submodules according to their needs for CPU time. Nonetheless, 28% of the total serial time is still spent simulating one unsplittable submodule, which limits our absolute speedup to 3.6. We reached a speedup of 3.07. This new partitioning requires a bit more work because the major hierarchical modules have to be split, and the whole structure must be reorganized. But someone familiar with the application can do this within twenty minutes, and even less with a GUI.

Further investigations about the discrepancies between the theoritically possible speedups and the real ones reveal that this application require many useless channel updates. This first means that many sychronizations of empty data are performed and then, indirectly, that the processes do not have a regular load all along the simulation. Thus, to further improve speedups, we may introduce disbalances in our regular pipeline test-application (see 3), so that we can study quantitatively their impacts on simulation time and investigate a dynamic load balancing at the granularity of a few δ-cycles rather than the static approach based on the global CPU load.

5 Conclusion

We demonstrate that it is possible to develop a parallel SystemC kernel with a user-friendly interface. An ideally balanced application shows that, despite the high level of synchronization required, speedups comparable to the number of computing nodes can be achieved. A performance model extracts figures about the CPU granularity with respect to synchronization needs in order to reach acceptable efficiency levels.

Even with a real-life coarse-grain application, where the CPU load is not equally distributed in space (over nodes) nor time (along the simulation), we showed that speedups close to their theoritical maximum can be achieved at very low development costs.

Our parallel SystemC kernel has still to be improved before being packaged for open source distribution. In particular the adaptation of existing optimistic PDES algorithms to avoid synchronization that are rarely required will be investigated.

References

1. OSCI `http://www.systemc.org/`: SystemC 2.0.1 Documentation: User's Guide, Functional Specifications, Language Reference Manual. (2002)
2. Martinelli, P., Wellig, A., Zory, J. In: IEEE International Workshop on Rapid System Prototyping. (2004) 193 – 200
3. Aboulhamid, E.M., et al.: eSYS.net (2004) `http://www.esys-net.org/`.
4. Meftali, S., et al.: SOAP based distributed simulation environment for System-on-Chip (SoC) design. In: Forum on Specification and Design Languages. (2005) To appear.
5. Savoiu, N., et al. In: Design, Automation and Test in Europe. (2002) 875–881
6. Ziegler, et al.: Theory of Modeling and Simulation - 2nd Edition. Academic Press (2000)
7. Skold, S., Ayani, R. Technical Report TRITA-IT R 94-19, Dept of Teleinformatics, Royal Institute of Technology, Stockholm (1992)
8. Misra, J. In: Computing Surveys. Volume 18. (1986) 39–65
9. Fujimoto, R.M. In: Communications of the ACM. Volume 33. (1990) 30–53
10. Ferscha, A.: Parallel and Distributed Simulation of Discrete-Event Simulations. In: Handbook of Parallel and Distributed Computing. McGraw-Hill (1995)
11. Naroska, E. In: Design, Automation and Test in Europe. (1998) 159–165
12. University of Cincinnati: SAVANT proj. (1999) `http://www.ececs.uc.edu/~paw/savant/`.
13. Cadwell, B., Browy, C. `http://www.avery-design.com/web/avery_hdlcon02.pdf` (2005)
14. Low, Y.H., et al. In: SIMULATION. Volume 72. (1999) 170–186
15. Fujimoto, R.M. ORSA Journal on Computing **5** (1993) 213–230
16. IST: MATRICE proj (2004) `http://www.ist-matrice.org/`.