Methodology and segmentation analysis for simulator parallelisation

Workshop on Simulation and EGSE for Space Programmes (SESP)
28-30 March 2017

ESA-ESTEC, Noordwijk, The Netherlands

Corentin ROSSIGNON [(1)], Nadie ROUSSE [(2)], Stéphan JAURÉ [(1)], Pierre VERHOYEN [(1)], Fernand QUARTIER [(1)]

[(1)] SPACEBEL
Ildefons Vandammestraat 7
Hoeilaart Office Park – Building B
1560 Hoeilaart, Belgium
Email : corentin.rossignon@spacebel.com, stephan.jaure@spacebel.com,
pierre.verhoyen@spacebel.com, fernand.quartier@spacebel.com

[(2)] Centre National d'Etudes Spatiales
18, Avenue Edouard Belin
31401 Toulouse Cedex 4, France
Email : nadie.rousse@cnes.fr

ABSTRACT

For operational simulators, average real-time performance is an absolute necessity; better performance is a significant bonus in testing and operational exercises. For hybrid simulators that include some hardware in the loop models, better than real-time performance and predictability are needed. The simulators have to cope with ever increasing requirements on accuracy, simulated computer speed, complexity and intelligent subsystems.

Studies on running several models in parallel have revealed several issue areas to address. With the current designs, more than half of the serial time is spent in the OBC (On Board Computer) emulator. The highest priority is to keep this critical path as deterministic and efficient as possible without introducing any latency potentially caused by synchronisations between the OBC and the other models. The lack of standard model interface for parallel execution is a limitation and solutions have to be developed. Moreover, our experience has shown that one should better have a complete model validation in a sequential manner before moving them to a parallel execution context as they might exhibit dependency problems and time inconsistencies. Further investigation revealed that the impact of parallelisation is significant, thus we developed a multi-scheduler approach that has been presented during the SESP 2015 event [1]. This approach can be used to speed up simulations by handling one sub-scheduler per thread.

While the above solution works well when simulators can be easily and manually separated, contrariwise there are several complications when a large series of small models have to be optimised. Firstly, their dependencies are indirectly expressed through their scheduling and their order in the scheduling lists. Therefore, as the causality between models has to be preserved, the effective dependencies become only known during runtime and some dependencies can be changed programmatically. Secondly, parallelisation of models that only consume a couple of microseconds might be counterproductive because of the synchronisation cost.

So, in a preparative first step, we have instrumented the simulator to maintain statistics (count, min, max, average and histogram) about each models execution time. We have developed a methodology to extract all dependencies from a simulator by analysing all connections of an assembled and scheduled simulator during a nominal run. With this data, we construct a graph in which each node represents an instance of a model and edges represent interactions between these instances. Then, we apply a graph partitioning method with weights on nodes corresponding to the computational time found during nominal run. Graph partitioning is a method often used in high performance computing for splitting work over several computer nodes. METIS [2] and Scotch [3] are two famous implementations of this method. For operational simulators, each partition tends to contain a set of tightly coupled models, often serving the same domain (thermal, dynamics, power …). A partition is a good candidate to assign instances to a sub-scheduler in the context of a multi-scheduler approach. This method meets our goal of assigning models over several sub-schedulers with a minimisation of the number of synchronisations between schedulers.

The expected results of the project are numerous. The tools allow to allocate models' instances to separate threads and to estimate the potential speed-up of multi-threaded simulation. Combined with the grouping of smaller models in one

single larger model and reducing sets of dependencies in one single segment, readability and understanding of complex simulators is significantly improved. It also adds an improved level of documentation and should improve the abstraction level of our introspection navigation tools. Finally, we come to the conclusion that the definition of model dependencies as a formal dependency tree at design level, which can later be compiled in a set of scheduling lists per thread, might be the best process in the long run.

Introduction

On the one hand, multi-core processors are common and could provide a great performance boost for heavy applications, on the other hand, current simulators are often single threaded and not made with parallelism in mind. So most of simulators do not benefit of the multi-core capability and are limited by one single core performance. This is excellent for simulators with relatively 'light" models, such as the Argos simulator that handles 500.000 events per second. In many cases, the simulation performance is not enough and some trade-offs are done on simulator representativity to maintain the needed performance.
Building a multi-threaded simulator is not as simple as setting a "use-multi-core" parameter during compilation, we need to specify how additional cores can help by offloading on them some works done by the main process. This is usually done with multi-threading paradigm and/or a runtime providing an abstraction of multi-core processors. We have already developed a multi-scheduler approach [1] in our simulation platform so that events can not only scheduled in one single scheduler but on multiple synchronised schedulers too. Then, a thread can be in charge of one or multiple schedulers, our abstraction layer only exposes schedulers.
Now that we have multiple schedulers at our disposal, we still need to assign events to these schedulers in order to exploit multiple processor cores. This distribution of events is the key to efficient parallelism. Later in this paper we will dissect this problem and provide a solution to it. The work presented here focuses on finding a methodology to transforming single threaded simulators into multi-threaded simulators.
A performance boost provided by multi-threading is profitable for testing, validating and qualifaction of new simulators; one can do more tests in less time. Parallelism is also a solution for increasing model accuracy; in operational simulators we need to guarantee that simulated time is at least real time.Without parallelism we are forced to limit the number of simulated components and/or doing some approximation about calculation intensive components.
Firstly, we enumerate the needed relevant data used for building simulators and useful data generated during a simulation run. Then, we study how to extract helpful parallelism-related information from these data. Next we show two direct applications of it. Finally, we discuss the current limitations of this approach.

Obtaining relevant data

The first step of our method consists to aggregate all available data about the simulator and its models. As starting point, we have the SMP2 models definition and connections between these models provided by four kinds of files: Catalog, Assembly, Configuration and Schedule. In our case, only Assembly and Schedule files are relevant.
Assembly files contain the definition of model instances, each of which composed of the name and the type of each model instance. These files are used to build the list of instances in the simulator. They also define the FieldLinks between these instances. The field link means that two instances have shared data and will need to be synchronized in a multi-threaded environment.
Schedule files contain definition of tasks and events. Tasks describe how instances will interact together and events define when tasks must be executed.
In addition to these configuration files, we also use files generated during run time. As a result of a nominal run of a simulator, we obtain two important files; the event log and the dump of the internal database. The first one contains a list of all executed events with the timestamp associated and the execution duration of the event. The second one, called dump base, is an ASCII file with all information known by the simulator about the models and connections between them. Of course, most of the data coming from dump base are redundant with SMP2 model definitions but dump base reflects the real state during runtime including the modification of connections done programmatically.

Segmentation analysis

Starting with all these pieces of data, we are able to build a graph of interactions between events. Each event is a node and the edges represent dependencies between events. Two events linked in the graph must be synchronized and cannot run simultaneously (Figure 1). Thanks to the event log file, we can attribute to every node a weight computed from execution time of the corresponding event.

A graph of events is just one of the many possible representations of the simulator, but this representation is convenient for future parallelisation. Building this graph is however not enough since in a multi-threaded simulation every time an event is triggered, one must define which thread will handle this event, and make sure that no other thread is executing related events.

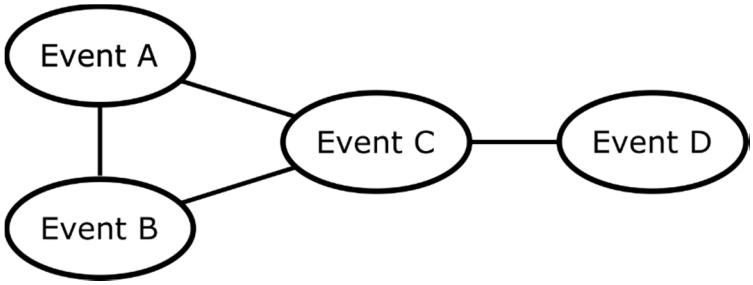

Figure 1 Example of event graph. During Event B execution, it is not safe to execute Event A or Event C but for Event D it is fine. In this example, execution of Event C should prevent other events to be executed in parallel.

We chose to segment the graph into parts, called partitions; a thread could handle multiple partitions but one partition can only be handled by exactly one thread. These partitions are built with the objective of reducing the number of edges between partitions and equilibrating the node weight. Edges can be seen as synchronisation points between threads; they are essential for maintaining temporal coherency during simulation.
The partitioning problem of a graph is NP-complete and it is often encountered in physical simulation. Some heuristics have been already developed, tested and provided under a software library. One of these famous libraries is SCOTCH [3] which is highly configurable but hard to use. Instead, we prefer use METIS [2], which is similar to SCOTCH, but provides a simpler interface.

Speed-up estimation

Parallelisation originates often from a speed concern. A speed-up is a ratio between the best sequential execution time and the parallel execution time. This value is linked to the number of processors (#NP), there are three different ranges. A speed-up between 0 and 1 means that we are losing time by using a parallel approach. The reasons vary from too much dependencies and synchronisations to false sharing line cache issues. But in the end, it is better to keep sequential execution when speed-up is lower than 1. A value between 1 and #NP means that parallel execution is profitable; the more the value is closer to #NP, the more the parallelisation is efficient. The last range is for values greater than #NP; this almost never happens and it often means that the caches are better exploited in parallel execution than in sequential execution, so there is some room for improving sequential execution.

The event log file contains all events executed by the simulator, each with with its start time and its. With this log, we can simulate the scheduling of all events in a multi-threaded environment. Each thread is represented by a timer and this timer is increased for each replayed event. Synchronisation between two threads is simulated by setting the last event with the biggest timer value. Event distribution is done with partitions computed in the previous chapter. We can assume that events from a partition will be executed by a unique thread; this is the most favorable case. Once all events have been replayed, we compare the biggest timer with the sequential time and we obtain an estimation of the best speed-up one can expect. This estimated speed-up is always belongs in the range [1, #NP] because cache properties and synchronisations cost are not taken into account.

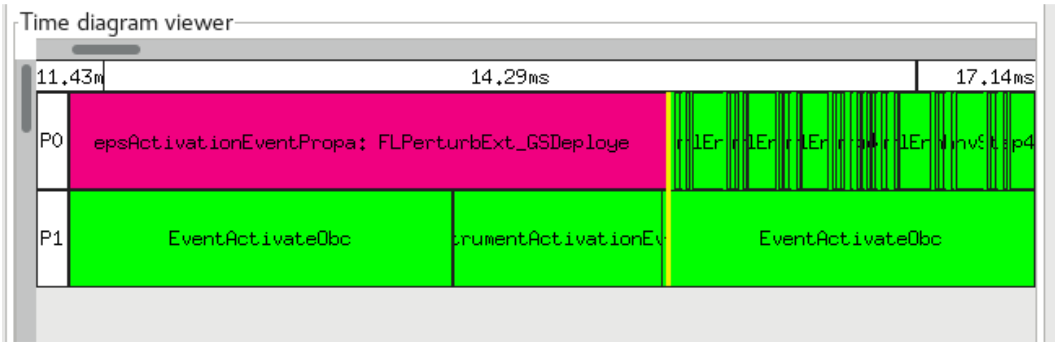

Figure 2 Diagram of a multi-threaded simulation

Our tool shows a time lapse diagram with a simulated parallel execution (Figure 2). There is one timeline per simulated scheduler. We can easily see which event is the most time consuming and, through the different colors, when an event is waiting for synchronisation.

Visualisation

We also developed a tool for displaying connections between events. This tool exports a graph in the DOT language, which is widely supported. Thanks to graph visualisation software such as graphviz [4], we have gained a new level of documentation for simulators. Graphviz computes the layout of the nodes using the different kinds of available layouts. One of them, called the "force-based layout" uses a physical simulation where edges act as springs. Nodes with a lot of edges become central and a visual segmentation of the simulator can be obtained (Figure 3). This is only one kind of layout, many others do exist and can be experimented with.

Unfortunately, graphs are difficult to read when it comes to the representation of a real-world simulator. There are a lot of small events which are not essential for the documentation but from the computational point of view, these events are perfectly legitimate. The next step for improving the visualisation is to add the possibility of creating groups of events with similar name and only display these groups. We are still working on an intuitive interface to do that more efficiently.

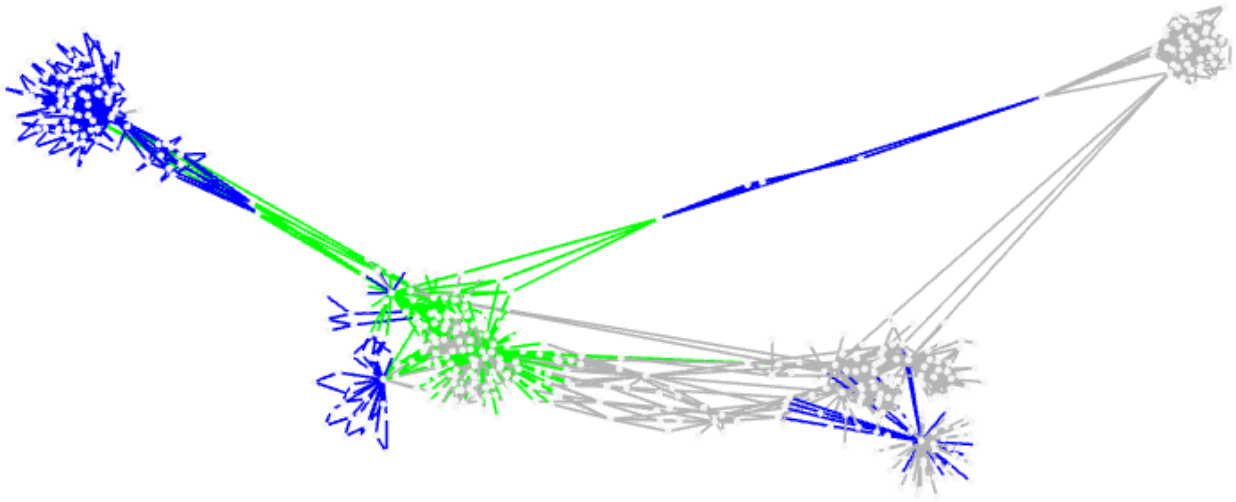

Figure 3 Example of an obtained graph with a force-based layout (Tulip software [5])

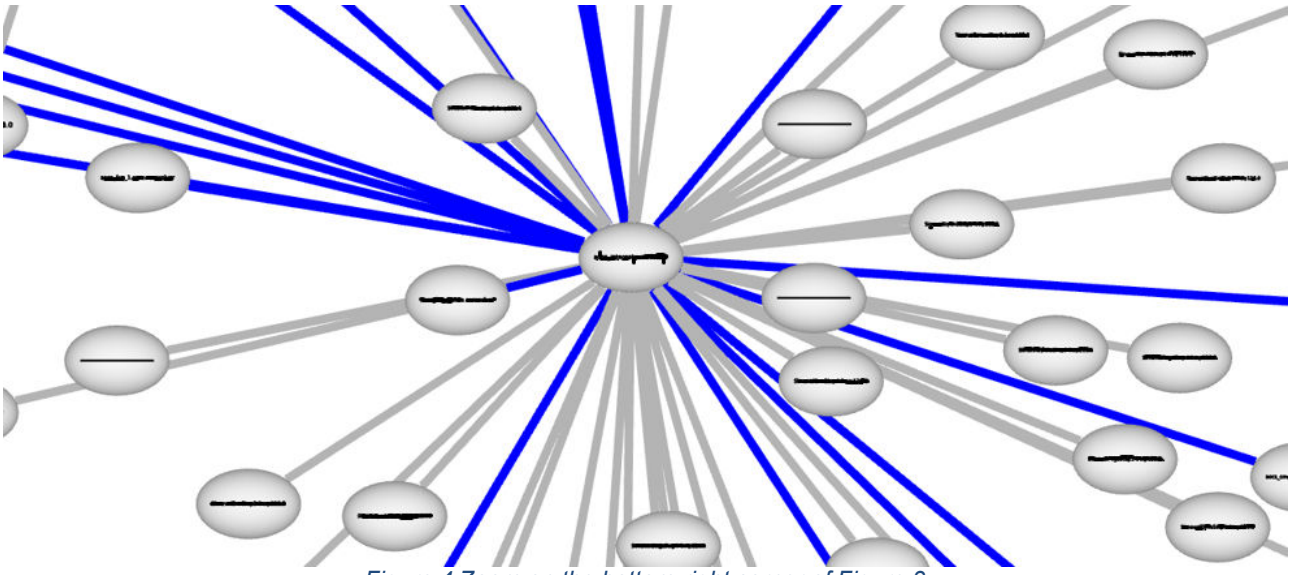

Figure 4 Zoom on the bottom-right corner of Figure 3

Limitations

In our analysis, so far, we almost never use the frequency parameter of cyclic events. We consider that connected events could always be executed at the same frequency and synchronisation is needed before each execution. In fact, typical operational simulators contain low frequency, high frequency and acyclic events. So, most of the time high frequency events don't interact with low frequency events but this information is not encoded in the graph. The most promising solution is to put a weight on edges because cutting low frequency interactions during graph partitioning is the better approach.

As current simulators are not designed with parallelism in mind, some assumptions are made but they are not necessary valid in a parallel environment. For example, in a discrete time event simulation, causality of events could be encoded with the order of insertion in the event list, but in a multithreaded simulation there is no absolute global order of insertion, it is relative to the current thread view of the events list. So, the causality can't be enforced by the event scheduler in a multithreaded environment in contrast to a single thread environment which can execute event in a FIFO fashion way. In one of our test cases, we found events with causality encoded like this and also, indirectly, in their names. For example, by insertion order, there are EventXX-step1, EventYY-step1, EventXX-step2 and EventYY-step2. It seems logical that, for a specific event type, step1 must be executed before step2. But, is it really important to execute first step of EventXX before EventYY-step1 (see Figure 5)? The answer to this question is crucial for parallelisation, if the answer is 'no', it means that EventXX-step1 and EventYY-step could be executed in parallel. This is a limitation of current simulator's description; we are losing useful information about dependencies because of the way we declare events and schedules at design time.

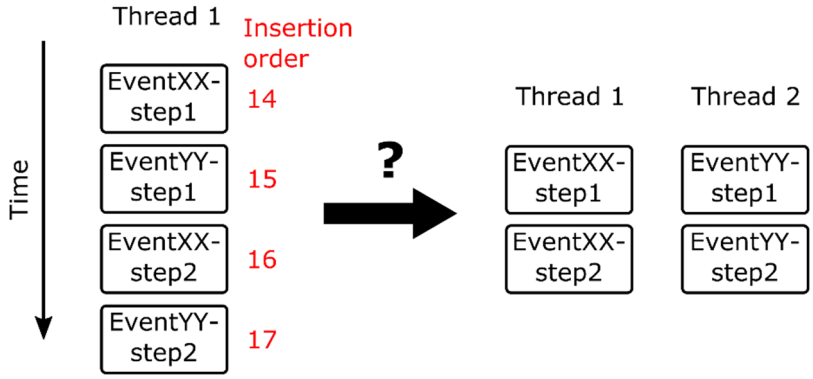

Figure 5 Example of dependancies encoded in insertion order. We lose the information about dependancy between EventXX and EventYY.

Another major problem concerns non-explicit shared variables, like global variables or file descriptors. As models could interact without using in-place mechanisms of the simulator framework, we could end up with a data race. When two threads are using the same variable simultaneously the result is undefined. Tools like ThreadSanitizer can help us to find data races in multithreaded programs during runtime but it is not enough to create safe multithreaded simulation. We need inputs from the integrator of models describing explicitly every shared variable not handled by the simulator framework.

Conclusion

During the last few years, increase of performance in new processors are mostly due to multiplication of computational cores, there is no more increase of the processor frequency. But, sequential programs cannot directly benefit of this improvement, they must use parallel paradigms, like multi-threads for example.
Unfortunately, current satellite simulators are not designed with parallelism constraints. They suppose using only one timeline with FIFO list of events. All configuration files are created for a single threaded simulator but they provide useful information about the simulator which can be reused for building a multithreaded simulator. Another useful source of information comes from logs provided during execution time, we can differentiate time consuming events from negligible events.
With all these pieces of information, we are able to build a graph representation of events and links between these events. Graph manipulation software, like Graphviz, yield another level of documentation for simulators. Several layouts are available and some provide useful visual segmentation of parts. Graphs can be partitioned with software graph tools with the goal of distributing event execution over threads. By replaying logs of events with distribution over threads, we obtain an estimation of future parallel execution performance.
Thanks to this methodology, we have a new level of documentation for our simulator and we can have a global view of event interactions. It also provides a good starting point for multithreaded simulation, events are separated in partitions with minimisation of interactions between them. Determinisme and causality should be preserved when switching to multithreading.
The next step is to use partitions in a real simulator environment and validate our segmentation analysis. There is also work in progress on improving graph visualisation and potentially exploit them for introspection tools.

References

[1] C. Lumet, N. Rousse and P. Verhoyen, “Multi-Scheduler and Multi-Thread: Possible With SMP2?,” *SESP,* 2015.

[2] G. Karypis and V. Kumar, “A Fast and Highly Quality Multilevel Scheme for Partitioning Irregular Graphs,” *SIAM Journal on Scientific Computing,* vol. 20, no. 1, pp. 359-392, 1999.

[3] C. Chevalier and F. Pellegrini, “PT-Scotch: A tool for efficient parallel graph ordering,” *Parallel computing,* vol. 34, no. 6, pp. 318-331, 2008.

[4] E. R. Gansner and S. C. North, “An open graph visualization system and its applications to software engineering,” *SOFTWARE - PRACTICE AND EXPERIENCE,* vol. 30, no. 11, pp. 1203-1233, 2000.

[5] D. Auber, D. Archambault, R. Bourqui, A. Lambert, M. Mathiaut, P. Mary, M. Delest, J. Dubois and G. Melançon, “The Tulip 3 Framework: A Scalable Software Library for Information Visualization Applications Based on Relational Data,” Inria, Talence, 2012.