Analyzing Data Dependencies for Increased Parallelism in Discrete Event Simulation

Mirko Stoffers*, Torsten Sehy*, James Gross‡, Klaus Wehrle*
*Communication and Distributed Systems, RWTH Aachen University
‡School of Electrical Engineering, KTH Royal Institute of Technology
stoffers@comsys.rwth-aachen.de, torsten.sehy@rwth-aachen.de,
james.gross@ee.kth.se, wehrle@comsys.rwth-aachen.de

ABSTRACT

To parallelize simulations, independent events have to be identified, which can be executed concurrently. The highest level of parallelism is achieved if the number of events identified as independent is maximized. Traditionally, this identification is based on time and location of events, only allowing parallelization if events on the same simulation entity are executed in timestamp order. To increase the level of parallelism, we propose a novel approach investigating another criterion for independence: If two events on the same simulation entity do not access the same data items in a conflicting manner, they can as well be executed in parallel. To this end, we propose static analysis of the model code for data access. To ease this process we develop the simulation language PSimLa similar to C++ but modified where necessary to increase analyzability without removing essential C++ features. First evaluation results show the potential of this approach and increase the confidence that data-dependency analysis can improve future parallel simulation.

Categories and Subject Descriptors

I.6.2 [**Simulation and Modeling**]: Simulation Languages

General Terms

Languages, Performance, Algorithms

Keywords

Parallel simulation; Static code analysis; Data dependencies

1. INTRODUCTION

The demand for simulation of more complex systems in higher degree of detail drives the need for parallel execution of simulation software. The parallelization gain primarily depends on the number of events identified as independent, commonly evaluated on time and location of events using the local causality constraint, which is fulfilled "if and only if each Logical Process (LP) processes events in nondecreasing timestamp order" [3, p. 32]. While this guarantees correct results [4], we argue that this is not a necessary condition. Two events can be independent even if their re-ordering violates the causality constraint, if they don't have data-dependencies. Today, there is only a single approach analyzing data-dependencies at compile time to increase parallel simulation performance [2]. However, this approach does not incorporate the challenging part of analyzing data access by pointers or references, hence it is restricted to a very small domain of models built without pointer or reference access. Hence, it is necessary to develop an approach applicable to Discrete Event Simulation (DES) models implemented in a structured language without removing essential features.

However, certain features of common general purpose languages like C++, especially pointers, do render data access tracking difficult to infeasible [1, 5]. Unfortunately, the huge set of Domain-Specific Languages (DSLs) for parallel simulation does not help solving this issue since neither of them is optimized for static analyzability. Instead, we propose a simulation language similar and compatible to C++, but aiming at increasing analyzability though not removing essential features without providing proper alternatives.

In this paper, we introduce the basic design of our language PSimLa, the current state of our proof-of-concept compiler implementation, and a first analysis approach. First evaluation results show that this approach is promising to speed up otherwise hard-to-parallelize simulation models.

2. THE PSimLa LANGUAGE

We design our DSL PSimLa for data-dependency based parallel simulation as a derivative of C++. To allow implementation of any DES model realizable in a general purpose language, PSimLa must be *Turing complete*. By *maximizing analyzability* the language design aids analyzing the code for data-dependencies. Under this constraint we *optimize execution performance* of the translated programs. To ease model development we *use well-established concepts* where possible and *maintain compatibility* to existing C++-code to enable step-by-step translation of preexisting models.

We base our language on C++ and the simulation elements of OMNeT++, and shape the compilation process in a way that PSimLa programs are first code-to-code translated into C++. During this step we also perform static analysis of PSimLa code for data-dependencies, which we then represent by additional C++ code. By compiling the output with a standard C++ compiler and running it with a modified version of OMNeT++, the provided dependency information can be used to gain additional speedup (see Sec. 3).

SIGSIM-PADS'15, June 10–12, 2015, London, United Kingdom.
ACM 978-1-4503-3583-6/15/06.
http://dx.doi.org/10.1145/2769458.2769487.

```
module MyMod {
  parameters:
    int myParam;
  gates:
    input myInputGate;
    output myOutputGate;
  private:
    int myInt;
    int myFn(int p1) { return p1+myParam; }
  protected:
    void initialize() { myInt=0; }
    void handleMessage(Message msg) {
      if(msg.getKind()==1) myInt=0;
      if(msg.getKind()==2) myInt=myFn(myInt);
      sendDelayed(msg,0,"myOutputGate");
    }
}
```

Figure 1: Example PSimLa Module.

For the language syntax, we adopt the building blocks and paradigms from OMNeT++. A *Module* can be defined similar to a class in C++. It is equivalent to a Simple Module in OMNeT++, hence developers need to implement an event handler and may provide an initialization and a teardown function. Additionally, PSimLa provides the standard syntax elements of C++ like primitive data types, classes, branches, and loops. However, the proof-of-concept implementation of our PSimLa compiler does not yet support every syntax element, but enough features are implemented to provide equivalent alternatives. For example, `for` loops can be replaced by `while` loops, changing neither semantics nor complexity. An example Module is depicted in Fig. 1.

The major difference between PSimLa and C++, however, is that PSimLa provides no pointer types, but only references. Under the hood, PSimLa translates references to C++11 smart pointers, enabling reference counting and deletion of objects at the end of their life cycles.

3. ANALYSIS TECHNIQUES

To investigate the analyzability of PSimLa we implemented a first analysis approach that aims at identifying data-dependencies between events. To this end, we assume that each data item can only be accessed by a single Module at a time. This is a common assumption in Parallel Discrete Event Simulation (PDES), e. g., when a simulation is decomposed into LPs where each LP can only access local data. However, while the local causality constraint [3] forces the events at each simulation entity to be executed in-order, our data-dependency information can help relaxing this constraint without changing simulation results. Hence, a Module can already process a future event even if another event with an earlier timestamp is executed later, finally eliminating too restrictive synchronization barriers.

Our static analysis works in five steps. 1. We identify and categorize events into different types. This allows us to store the derived dependencies on an event type basis, as the concrete event instances are not known at compile time. 2. We track, which data items are accessed by events of the different types. Since different events of the same type might access different data items and the Turing-completeness of PSimLa does not allow to reliably detect, for example, which branch the program flow will take at runtime, we chose to conservatively overestimate the data accesses. This means that two independent events might not be executed in parallel if we cannot guarantee this independence based on the information gathered. 3. We determine the scheduling relations, i. e., which other events an event handler will schedule at runtime. This is important to avoid conflicts with those events if another event is executed early. 4. We infer the event dependencies and store which event types depend on which other event types. To this end, we generate C++-code that allows determining the type of a given event instance and deriving its dependencies. 5. At runtime, we access this information to yield decisions whether the next event can be safely executed in parallel immediately, even if the local causality constraint could not guarantee correctness.

4. EVALUATION AND CONCLUSION

We performed first evaluations of the analysis approach. Synthetic benchmarks of simple events only performing read operations – which where previously not parallelizable due to causal violations – show almost linear speedup on a 12-core simulation platform, while the same model with write operations cannot be speeded up. This confirms that the approach correctly identifies event dependencies and independencies in these scenarios. A case study of a 57 node Wireless Mesh Network, which is speeded up by traditional parallel simulation only by a factor of 3 to 4, can as well gain close-to-linear speedup by our data-dependency analysis.

We conclude that data-dependency analysis is a promising technique for future improvements on parallel simulation. To this end, it does not suffice to account only for member variables of modules, but data items accessed by pointers or references have to be incorporated into the analysis to include a wide set of simulation models. To circumvent the infeasible problem of pointer analysis [1, 5], a language designed with analyzability in mind, like PSimLa, can aid the analysis without restricting the opportunities necessary for model development. Future efforts in enhancing the PSimLa compiler implementation and the analysis approach allow investigation of the feasibility of data-dependency analysis on a broad range of simulation models.

Acknowledgments

This work has been co-funded by the German Research Foundation (DFG) within the Collaborative Research Center (CRC) 1053 – MAKI.

5. REFERENCES

[1] D. Binkley. Source Code Analysis: A Road Map. In *Proc. of Future of Software Engineering*, (Minneapolis, MN, May 23–25, 2007) IEEE, Los Alamitos, CA, 104–119, 2007.

[2] W. Chen, X. Han, and R. Dömer. Out-of-Order Parallel Simulation for ESL Design. In *Proc. of the 2012 Conf. on Design, Automation & Test in Eur.*, (Dresden, Germany, March 12–16, 2012) IEEE, Los Alamitos, CA, 141–146, 2012.

[3] R. Fujimoto. Parallel Discrete Event Simulation. *Communications of the ACM*, 33(10):30–53, 1990.

[4] R. Fujimoto. Parallel and Distributed Simulation. In *Proc. of the 31st Winter Sim. Conf.*, (Phoenix, AZ, Dec. 5–8, 1999) ACM, New York, NY, 122–131, 1999.

[5] M. Hind. Pointer Analysis: Haven't We Solved This Problem Yet? In *Proc. of the 2001 ACM Workshop on Prog. Analysis for SW Tools and Eng.*, (Snowbird, UT, June 18–19, 2001) ACM, New York, NY, 54–61, 2001.