Accelerating the Simulation of Wireless Communication Protocols using Asynchronous Parallelism

Dominik S. Buse
Paderborn University and TU Berlin
Germany
buse@ccs-labs.org

Georg Echterling
Paderborn University
Germany
georg@echterling.net

Falko Dressler
TU Berlin
Germany
dressler@ccs-labs.org

ABSTRACT

Simulation is a key tool in the development of wireless systems, protocols, and applications. This is especially true for large and mobile networks such as Vehicular Ad-hoc Networks (VANETs), as real-world experiments quickly become too expensive and complex with increasing numbers of nodes. However, accurate simulation of such wireless networks may take a long time to compute. This is mainly due to the sequential processing in event-based simulation cores like OMNeT++ or ns-3. This is especially troubling when debugging new protocols or when interfacing with Hardware in the Loop (HIL) or other real-time simulation. In this paper, we propose a new approach to accelerate simulation of wireless communication by utilizing asynchronous background computations. Expensive computations that can be isolated from the simulation kernel are started as early as possible and pushed to background threads. At the point in time when the result is needed, the computation may already be complete. This allows for parallelization without an explicit lookahead value. As a proof-of-concept, we implemented the concept in Veins, a state-of-the-art VANET simulator. By offloading wireless signal attenuation model computation to the background, we achieve speedups of up to 3.5, depending on the scenario.

CCS CONCEPTS

• **Networks** → **Network simulations**; • **Computing methodologies** → *Discrete-event simulation*; • **Theory of computation** → *Parallel computing models.*

KEYWORDS

Parallel simulation, wireless network simulation, asynchronous parallelism, vehicular networking

ACM Reference Format:
Dominik S. Buse, Georg Echterling, and Falko Dressler. 2021. Accelerating the Simulation of Wireless Communication Protocols using Asynchronous Parallelism. In *Proceedings of the 24th ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM '21), November 22–26, 2021, Alicante, Spain.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3479239.3485683

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MSWiM '21, November 22–26, 2021, Alicante, Spain
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-9077-4/21/11…$15.00
https://doi.org/10.1145/3479239.3485683

1 INTRODUCTION

Wireless mobile networks have become a ubiquitous pillar of our modern society. Thus, significant effort is spent on the development of wireless systems, protocols, and applications. Along these lines, simulation is one of the most important tools supporting the development and the evaluation of wireless communication protocols [19]. Therefore, powerful and efficient simulation tools are of great importance for the field. Without loss of generality, we concentrate on one application field, which, from a simulation perspective, is one of the most challenging, namely Vehicular Ad-hoc Networks (VANETs). VANETs typically feature mobility of the communicating nodes, large scale, and high complexity of the wireless communication environment. They thus require either very abstract modeling or substantial compute power to simulate [15].

As most simulators for wireless communication are implemented as single-threaded sequential programs, they take a long time to compute. Of course there is a whole field of research on parallel and distributed simulation [9]. While there are simulators implemented in a parallel fashion [32], widespread network frameworks like OMNeT++/INET, ns-3, or the vehicular networking simulator Veins do not provide parallel simulation for wireless networks. We see two major reasons: First, having a shared wireless channel in the simulation makes existing approaches to parallel simulation much less effective. This is because the structure of wireless mobile networks is more like a tight mesh with many local connections instead of the scale free graph many wired networks form. Logical Processes (LPs) synchronized via conservative protocols require some guaranteed lookahead time, which is easy to determine in wired networks. But it becomes much shorter for a network of wireless nodes that are all potentially connected, close by (low propagation delay), and can send at almost any time. And due to the mobility, partitioning the network becomes much harder as well. Optimistic coordination, e.g., the *TimeWarp* algorithm [14], would probably need to perform a lot of rollbacks as many wireless nodes could influence each other. Second, the simulations need to be deterministic to be reproducible and to ease the debugging process. Randomness in the simulation models is usually realized using Pseudorandom Number Generators (PRNGs) and a stable seed value. Now, the interconnections may be subtle; just drawing one fewer number from a PRNG can disturb the whole rest of the simulation by causing a "butterfly effect".

The problem described is not very severe when running parameter studies. Here, all CPU cores can be fully utilized, each for a single simulation run, i.e., exploiting process-level parallelism. But most of the time, the simulation or protocol will be under development or even debugging. And this means that the developer will have to wait for the single-threaded simulation to run to completion or the place where a bug happens, possibly many times over. The problem

gets even worse if debug builds have to be used, which run even slower due to disabled compiler optimizations. So in these cases, the developer is looking for low delay instead of high throughput. A very similar issue arises if the simulation is coupled to a real-time system. Setups like Hardware in the Loop (HIL) are becoming widespread for component testing and the likes [7, 13]. Similar environments include human in the loop approaches such as the Virtual Cycling Environment (VCE) connecting human interaction with discrete event simulators (DESs) [30]. Again, running multiple separate simulation processes does not help. Only one result is needed, but within tight timing boundaries, which limits the size of the simulated system.

In order to speed up wireless simulations despite all these issues, we propose a different way to parallelize the simulation to multiple CPU cores. We side-step the synchronization problem by decoupling computationally expensive parts of the simulation and processing them asynchronously in background threads. This is the basis for our concept, because it can exploit the resulting parallelism without being constrained by (small) lookahead values. It provides speedup without actually parallelizing the core of the simulation, making it much easier to adopt for users. What we need to find are isolated computations with a clear interface, like hot loops or costly algorithms, which could be decoupled from the simulation kernel or are already provided by separate library. They should take some input, do costly computation, and return results relevant to the later simulation. Ideally, the data needed to perform the computation should be available in one event, while the result they produce is only needed at a later event—even if only by a tiny amount of simulation time. Because then, they could be moved out of the main simulation thread, processed in the background, and may finish while the main thread continues the simulation.

In this paper, we show how a simulation of mobile wireless networks can be accelerated. We describe the concept of asynchronous background computation in discrete event simulation and the requirements for it to work. To complement the theoretical concepts, we showcase each step using the vehicular networking simulator Veins [28] as an example. We show how we identified the computation of signal attenuation models as the largest consumer of computation time; we describe how we adapted the models to run isolated from the simulation kernel and the implementation of the asynchronous background computation; and we verify the correctness and evaluate the performance of the resulting parallel simulation using two typical scenarios for VANET simulations.

Our extension requires no changes or even awareness by most users, as they may not be experienced in parallel simulation. Also, no change to the simulation kernel is needed, thus, the system is easy to adapt to other simulators. We will of course make our system available as open source to the research community.

Our main contribution can be summarized as follows:

- We developed a technique to accelerate the simulation of wireless communication protocols through asynchronous parallel processing,
- we present an implementation accelerating the signal attenuation models of Veins as a proof of concept,
- we evaluated the speedup of our implementation for speed, debug builds, and real-time applications.

2 RELATED WORK

The typical simulation tool sets used for the simulation of wireless mobile networks have some kind of parallel distributed simulation support. However, these are mostly traditional parallelization and distribution approaches focused on point-to-point communication, using Logical Processes (LPs) and conservative synchronization with lookahead times. E.g., OMNeT++ has supported the null message algorithm for a long time [26]. Its implementation uses the Message Passing Interface (MPI) and utilizes link delays as lookahead. Similar approaches have been implemented for ns-3 [23], also being built on top of MPI. While scaling for wired networks has shown impressive speedups, "a distributed simulation in ns-3 requires at least one point-to point link within the topology" [23].

Pioneering work for parallel simulation of wireless systems has been done in in [24]. It also uses LPs with conservative synchronization via lookahead windows, combined with a geographically partitioned simulation. The key advantage proposed by the authors is to derive "large" lookahead windows from the properties of the simulated protocol. I.e., incorporating inter frame spaces, backoff intervals, etc. into the lookahead computation. However, this binds the acceleration of the simulation model to a specific protocol and may lead to problems with heterogeneous communication. Also, the speedup may decrease with more dense and mobile networks, e.g., when simulating VANETs on a motorway.

A recent study [1] surveyed approaches for ns-3 in order to simulate an LTE network for public safety applications. This means large-scale networks and the option to emulate parts of the system in real time, similar to a HIL system. The authors conclude that building an LTE network simulation for public safety applications would require significant extra work for the three most common approaches in the literature: MPI based solutions depend on point-to-point connections and lookahead values, while multithreading and graphics processing unit (GPU)-based approaches require highly complex changes to the core of simulation models.

The same authors later implemented a large-scale LTE simulator by coupling ns-3 and CORE [25]. Their simulator uses the MPI protocol and is able to emulate an LTE network in real-time to perform HIL studies. Parallelization is done using LPs, each representing an LTE eNB in the simulation and with point-to-point links of known delay between them. A static lookahead value given by the model designer is used, which has to be shorter than the time between two dependent events, with a value of 1 ms in the paper. While this may suffice for eNBs with point-to-point links between them, it is not a feasible order of magnitude for VANET simulation.

The *HORIZON* [17, 18] extension to OMNeT++ accelerates simulations by identifying events that are independent of each other and running them in parallel. It does so by extending (at least a subset of) the events from instantaneous points in time to time intervals by adding a duration. Events may only change the simulation state after their duration has passed. Thus, events with overlapping time intervals can not depend upon each other—they either would not have been allowed to start or would not be able to use each other's results. So overlapping events can be scheduled to run in parallel. However, it requires significant changes to the simulation kernel and assignment of a duration to a significant portion of the events, which complicates simulation model design. The evaluation also

shows that the speedup depends highly on the duration of events, with less speedup for events of shorter simulation duration and more computational complexity. The lowest duration analyzed in the original paper is 1 µs, roughly the propagation delay of a signal across 300 m—much more than many messages in VANETs travel.

A different approach is the parallelization of individual component models in the simulation, e.g., building obstacle shadowing [20]. This allows to speed up selected computationally expensive portions of the simulation while leaving the rest of the model untouched, yielding some speedup at low cost to the user. However, as the main simulation thread still waits for the completion of the parallelized computation, it leaves potential speedup on the table.

For VANET simulation with Veins specifically, a distributed simulation scheme based on the High Level Architecture (HLA) has been proposed [10]. Coordinated by HLA federates, multiple instances of Veins each cover a portion of a Manhattan grid scenario. Vehicles passing from one region into the next are transferred to the control of the respective region. In contrast, there is no notion of exchanging simulated wireless messages between the instances of Veins, meaning the reception of vehicles close to region borders may vary with the simulation layout. So results of simulations with different numbers of Veins instances will differ and may be inaccurate due to the potential lack received messages or interference.

There is currently ongoing work to extend ns-3 into the Spectrum Sharing Simulator (S3) [2]. The goal is to be able to run large-scale, combined LTE, LTE-Advanced, and 5G-New Radio networks to analyze effects of spectrum sharing. A combination of optimistic synchronization using a compiler-assisted rollback mechanism and conservative synchronization using lookahead values provided by the model designer shall be used for parallelization. Though at the time of writing of this paper, said work has not yet concluded.

COSIDIA [22] is another new parallel simulator for wireless mobile systems, specifically VANETs, with a focus on real-time capability for HIL testing. Similar to *HORIZON*, events are extended into actions with a start and end time and then distributed to lightweight fibers, which implement LPs, for parallel execution. *COSIDIA* promises "fully deterministic functional behaviour when no external hardware is attached" and aims to execute all events within defined real-time boundaries. However, *COSIDIA* is still in an early stage of development, so no conclusions of the performance when simulating realistic scenarios and protocols can be drawn yet.

Aside from parallel and distributed acceleration techniques, there have been attempts to improve the (single-core) performance of Veins. Since forking off from MiXiM, Veins has used a *maximum interference distance* to limit the amount of stations needing to be checked to be potential receivers of a packet. When a message is sent by pushing copies of the message to potential receivers, only stations within that distance from the sender are considered. This can significantly speed up the simulation by reducing the computational complexity by orders of magnitude. However, it can potentially change the outcome of the simulation in some cases. Even very low interference levels of multiple messages may still add up and influence the decision of whether some other message can be successfully received or not. Originally, the maximum interference distance was computed from the signal characteristics in Veins. But with the introduction of antenna models [8] in Veins 4.5, the distance has to be configured by the simulation author.

More recently, a re-work of the signal implementation of Veins in version 5.0 introduced improvements to the code performance and the concept of *thresholding* [4]. With thresholding, the computation of signal attenuation models may be aborted early if the attenuated signal falls below a given threshold before all models have been computed. This can reduce the computational complexity and thus speed up the simulation itself without changing the simulation outcome. Attenuation models that may increase the received signal strength (e.g., two-ray interference) have to be evaluated completely before thresholding can be applied. It works best if more complex models are computed later in the chain of attenuation models, as skipping them provides the largest speedup. However, this optimization provides very different speed-ups depending on the channel models involved and only speeds up this part of the simulation.

3 ASYNCHRONOUS BACKGROUND COMPUTATION

Typical DESs like OMNeT++ or ns-3 work by processing events in discrete points in time. Scheduled events are stored in a future event list (FEL) and ordered by their time stamp. Once the simulator finishes processing an event, the next one is fetched from the FEL. Time intervals between the finished and next event are simply skipped. Events can schedule new events themselves, which are sorted into the FEL at the appropriate time. Once there are no more events left or some other end condition is met, the simulation stops.

While events usually represent a discrete point in time for isolated entities in the simulation, there is often a semantic relationship between them (c.f. Figure 1). E.g., the event of the physical layer of a node starting to broadcast a wireless message and the resulting event at the point in time where of said beacon first reaches a potential receiver. However, this semantic relationship is typically not encoded into the simulation directly. Instead, it is part of the design of the simulation model. Additionally, there is often some degree of freedom in how what part of the model gets implemented in which event. Following the previous example, the attenuation of the signal of the broadcast could be computed in in the sending event or at the receiving event.

Typically, there is no link between the simulation time, i.e., the time in the simulation, and the wall clock time of the computer running the simulation software. The progression of simulated time may vary drastically: Multiple close-by events with high computational complexity may take a long wall clock time to compute but only advance the simulation time a little (cf. Figure 1). E.g., the computation of signal attenuation models for multiple copies of a broadcast, with only the difference in propagation delay between them. Meanwhile, large gaps between events can lead to fast progression of simulation time without much computation time spent. E.g., the time between two broadcasts.

These two facts can be exploited to speed up the simulation by offloading expensive computations between two events from the simulation main thread to some background (worker) thread. Assume the result of an expensive computation is needed in event e_5, as shown in the bottom half of Figure 1. Normally, this result would be computed in event e_5, making it take a long time to compute. But the computation could already be performed in event e_2 which occurs at an earlier point in simulation time ($t(e_2) \leq t(e_5)$).

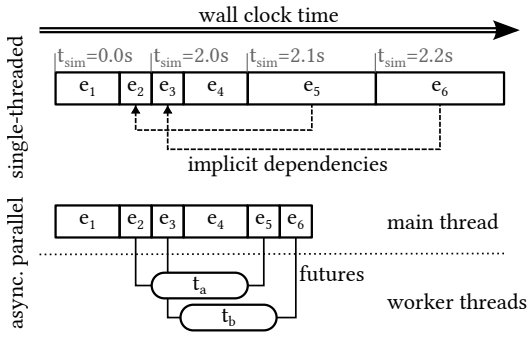

Figure 1: Time progression in single-threaded (top) and asynchronous parallel (bottom) simulations

There may be an arbitrary amount of simulated time and other events (even 0) between e_2 and e_5. The computation can already be started in event e_2, but performed asynchronously in a background thread in parallel to the main thread. The main thread can continue processing further events, as the result of the computation is not yet needed. Once the main thread reaches e_5, it fetches the result of the computation from the background thread. This may involve waiting if the computation is not yet fully finished, but it will typically take less time than doing the whole computation in the main thread. Then, the computation result can be used and the simulation continues as usual. Naturally, there can be more than one background computation active in parallel, though the actual number of computations being processed in parallel depends on the simulation hardware. This procedure can speed up the overall simulation as the main thread can go through the FEL faster while other threads and cores take care of the expensive computations. Having many background tasks scheduled at the same time makes the procedure even more efficent, as work can be distributed to more cores. Tasks will only start queueing up if all cores are utilized and the system runs at peak efficiency.

While this is similar to other offloading techniques, e.g., to a GPU, it exploits the nature of the DES by advancing the simulation while the offloaded tasks are being processed. Also, the implementation of the model and the simulation core itself did not have to be modified or re-implemented, like it would have been for a GPU. Only a little instrumentation is needed to offload computation tasks to a background thread and obtain their result later. A typical way to implement this is using Futures and Promises [11]. When the task is offloaded, the main thread receives a future instead of the actual result (c.f. Figure 1). The future can be used to await the completion of the offloaded task and obtain the actual result once it is available, with no further need for synchronization primitives. The background thread receives the promise, through which it communicates the completion of the computation and its result. The main thread can attach the future to the event or simulation object that will eventually request the result of the computation.

However, there are limitations to this asynchronous offloading approach. The offloading task may not interfere with the simulation progression itself, e.g., by scheduling events. Such actions may lead to unrepeatable and non-deterministic behavior, as the point in simulation time at which the background thread accesses the simulation core is not defined. In addition, direct interaction with other objects in the simulation would require locking to avoid race conditions. Thus, background tasks should ideally be pure functions without any side effects and with a set of input parameters fully determined at the point at which they get offloaded. This includes less obvious inputs such as random numbers drawn from a PRNG. As there may be multiple background tasks being processed at the same time in different threads, the order of the calls to the PRNG could become non-deterministic. This could lead to non-reproducible changes in the simulation outcome, which have to be avoided. Even reading non-constant data from other simulation objects, e.g., the location of a mobile node, is prohibited as the node may or may not have moved in the main thread at the time.

In practice, this often means moving the start of the computation to the point in simulation time at which they first become possible. It may even be useful to include processing delays of the system represented by the model to create more opportunities for the completion of background tasks while the main thread processes other events. The kind of computation that gets offloaded is flexible and up to the implementer. This may be common simulation elements like signal attenuation models, calls to external tools, or even computation-intensive application layers in the simulation.

4 APPLICATION TO VEINS

To demonstrate our asynchronous background computing concept, we implemented it in Veins [28], a state of the art VANET simulator. We based our work on the most current release: version 5.1.

4.1 Profiling: What to Parallelize

With the concept of asynchronous background computation in place, we now need to find portions of the Veins code to accelerate with it. These portions need to have two qualities: easy to isolate from the simulation kernel and computationally expensive enough to provide a significant speedup. We profiled the code of Veins 5.1 on the motorway scenario later used for the evaluation (see Section 5.1) using the `perf` sampling profiler. As long as the simulation runs, `perf` samples the currently active stack frame of the program at regular intervals and saves these samples to a profile on the disk.

The profile shows that two call trees take up the largest share of the computation (see Table 2 for more details): `processNewSignal` with 43.1 % and `processSignalEnd` with 46.1 % of the samples. Both are functions of the `Decider80211p` module and are responsible for the decision of signal detectability on reception start and decodability on reception end, respectively. While the functions do work of their own and call several other functions, both spend most of the time with the evaluation of signal attenuation models.

These signal attenuation models are a good fit for asynchronous background computation. They can be implemented as side effect-free functions that produce a single result, the attenuated signal. All data they need for their computation is well specified and can be collected ahead of time. So they do not need to query the simulation for further information. Finally, they are plain computations and do not schedule any events of their own. In addition, signal attenuation is present in virtually all simulations using Veins. So speeding them up may provide a great benefit for many users. We thus decided to parallelize them using asynchronous background computation.

4.2 Wireless Message Sending in Veins 5.1

In Veins 5.1, message sending and signal propagation are modeled as follows: Each host in the simulation that is capable of wireless communication contains at least one Network Interface Card (NIC). The NIC contains the implementations of the Medium Access Control (MAC) and physical (*PHY*) layers of the respective stack. By default, this is the IEEE 802.11p vehicular networking stack, but others are available through extensions and the concept is the same for all of them. The implementation of the MAC and PHY layer uses hierarchies of modules from the OMNeT++ library.

At the top of the PHY layer hierarchy lies the `ChannelAccess` module, which takes care of disseminating messages on the wireless channel. Whenever such a wireless message—named `AirFrame` in Veins—is sent, the `ChannelAccess` creates a copy of the `AirFrame` for each potential receiver and sends that `AirFrame` copy to it. Potential receivers are all other NICs within a *maximum interference distance*, which is specified by the simulation designer (see Section 2 for details). Each `AirFrame` contains a `Signal`, representing the PHY properties of the signal by which the message is transmitted via the wireless channel, e.g., timing, frequencies, and power levels [4]. The `Signal` also contains information about the sender and receiver of the `AirFrame` as well as a list of attenuation models that are used to compute the power levels at the receiving NIC. Ultimately, each `AirFrame` copy is sent by scheduling a reception start event at the receiver with an offset in time that represents the propagation delay between sender and receiver (c.f. Figure 2).

At a reception start event, the receiver examines the received AirFrame. First, the receiver checks if the received signal is strong enough to be detectable. This triggers the evaluation of the attenuation models of the `Signal` to compute the received power levels (including antenna gains [8]). If *thresholding* [4] is enabled, the evaluation of attenuation models may be aborted early if the resulting power levels are below the detection threshold. Otherwise, all attenuation models are evaluated. Afterwards, the receiver decides whether it can tune in on the signal, considering the power levels and its own current state (e.g., transmitting or receiving another message). In any case, the received `AirFrame` is stored as a potential interferer to other received signals and an event for the reception end is scheduled based on the length of the message.

At the reception end event, the receiver checks if the received `AirFrame` can be successfully decoded by computing its Signal-to-Interference-plus-Noise Ratio (SINR). To do so, the receiver collects all other received signals overlapping with the current one in question. If thresholding is enabled, this may trigger the evaluation of previously skipped attenuation models of both interferers and the current one, in order to determine their final power levels. With the SINR, the receiver can apply the bit error model and finally decide whether the message can be decoded—also considering collisions and other errors. At success, the `AirFrame` gets moved up the stack to the MAC layer. Otherwise, it is deleted—though a reference may be kept until it may no longer be an interferer for other frames.

4.3 Asynchronous Parallelism in Veins

With asynchronous background processing, the start of the evaluation of attenuation models is moved from the reception to the sending event, as show in Figure 1. To enable offloading of the attenuation model evaluation, the implementation needed to be partially restructured. As explained in Section 3, all data needed for the evaluation needs to be available before the background task is spawned. No further access to the running simulation may be performed by background tasks. There also needs to be an interface to schedule received signal copies for background computation and await the completion of tasks. This interface should accept a collection of received signal copies, as all of them are created at the same time and each scheduled tasks incurs some overhead.

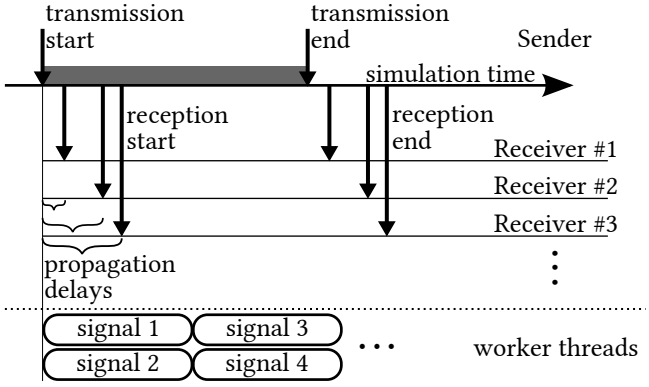

Figure 2: Timeline of an AirFrame with the start of the attenuation model evaluation

We solved these requirements by gathering all received signal copies into a `SignalGroup` object within the `ChannelAccess` module. The `SignalGroup` takes care of storing the parts of a `Signal` that are the same for each copy, e.g., sender information, transmission time or message duration. It also efficiently stores the individual data per receiver, e.g., the receiver information, propagation delays, attenuated power levels, and applied attenuation models. Upon its construction, the (modified) instances of attenuation models are created and collect all data needed for later evaluation, though their evaluation is not yet performed. This allows each attenuation model class to collect the necessary data through polymorphism. It may require taking snapshots of the simulation state, e.g., of positions of other vehicles for vehicle obstacle shadowing. If an attenuation model requires random numbers, they also are drawn at this point to guarantee reproducibility through deterministic access to the PRNG. After construction, the `SignalGroup` is considered read-only to the simulation kernel, while only the background computation tasks are allowed to modify it. Each `AirFrame` copy contains a `SignalInstance` through which the receiving NIC can access its signal without explicit knowledge of the `SignalGroup`. This replaces the `Signal` class of previous Veins versions.

Each `SignalGroup` contains multiple (often tens or even hundreds) of received signal copies, each with a set of attenuation models to evaluatate. Scheduling the computation of each received signal copy as a separate job wrapped in its own future provides the highest potential to run in parallel, but also incurs the highest overhead. Instead, received signal copies can be grouped and scheduled as a chunk, which reduces overhead but may leave available background threads idle and thus reduce throughput and distort real-time performance. The most efficient solution to this problem depends on the hardware platform running the simulation.

Thus, hard-coding may not fit all situations. So we introduced an abstraction named a `ParallelStrategy` which decides how received signal copies are bundled and made it runtime configurable. Each `SignalGroup` sent to the wireless channel gets passed to the configured `ParallelStrategy`. It then schedules bundles of received signal copies to be computed on the background threads and provides an interface to wait for and retrieve the results for each `SignalInstance`.

We implemented and evaluated four `ParallelStrategy` types:

- *separate* schedules each received signal copy as its own background task, aiming for maximum parallelization.
- *chunked[N]* bundles N received signal copies into one chunk and schedules each chunk as a background task with the goal to reduce overhead.
- *hybrid[N]* schedules the first N signal copies (whose results will be needed earliest) as separate background tasks. The rest are bundled into one future. This aims to provide fast results for soon-needed results but keep the overhead low.
- *sync* computes all models in the main thread synchronously, as soon as the signal gets sent out. It is a special case intended for debugging, single-core runs within process-parallel studies, and for a more direct comparison with Veins 5.1.

To actually schedule the background tasks, we used the *TaskFlow* library [12] in version 2.7 (to only require a C++14 compliant compiler, like Veins 5.1 does). It provides an efficient thread pool implementation with work stealing and supports future objects compatible with the C++ standard library. Thanks to its permissive MIT open source license and header-only implementation, TaskFlow can easily be distributed with Veins. In preliminary tests, we also tried out the `std::async` function of the C++ standard library to schedule background tasks. But we had to discarded it due to its low and unpredictable performance.

Once a background thread starts processing a received signal, it first applies the antenna patterns to the signal and then starts to evaluate the list of attenuation models. After this task is finished, a future object may be fulfilled by the TaskFlow library—depending on the `ParallelStrategy`. If thresholding is enabled, this may still abort early if the remaining signal power level is low enough. However, this may make it necessary to run the skipped attenuation models later when interferer power levels are computed. This currently has to happen on the main thread, as it is not predictable which received signals may be relevant interferers to other signals.

Some further changes were made to Veins in order to improve the performance and reduce the overhead on the main thread.

- The `Coord` class no longer inherits from the `cObject` class hierarchy to make it a plain struct that can be safely passed to background processes and is cheaper to allocate and free.
- The implementation of the spectrum has been changed from an array of frequencies per `Signal` instance to a shared pointer to a pre-allocated spectrum class. This significantly reduces the amount of data that has to be copied for each signal instance.
- Instances of attenuation models are now shared across the simulation and not duplicated per NIC. Only a thin wrapper responsible for data collection remains for each signal instance.

4.4 Result verification

Most of the changes described above, including the background processing, do not alter the outcome of the simulation in any way. This was extensively tested and verified using the *fingerprint* mechanism of OMNeT++. However, as data needed for attenuation model evaluation now needs to be collected at the time of sending the signal instead of the time of reception, some small changes may be introduced. This is the case for the vehicle obstacle shadowing attenuation model. If the point in time of the collection of vehicle positions moves from after vehicle updates are received from TraCI (in the old code, at signal reception time) to slightly before said vehicle update (in the new parallel code, at sending time), the collected vehicle positions may differ. This in turn can lead to different outcomes of the signal attenuation, e.g., if a vehicle is now in the line of sight of a transmission which reduces the received power level of the signal. And this may lead to further propagation of changes, as the different received power level may lead to more or less random numbers being drawn in the bit error model. Thus, the whole outcome of the simulation from that point on may be slightly different. We could however establish that this effect is acceptable: First, the change is only observable when comparing Veins 5.1 with our parallelized version—the modified code always behaves deterministically, regardless of the chosen hardware platform, number of threads, or `ParallelStrategy`. Second, using statistical evaluations and analysis of the simulation event log (not shown in this paper), we found that the resulting change is similar to choosing a different random seed for Veins.

5 EVALUATION

To showcase the speedup possible in Veins with asynchronous background processing of signal attenuation models, we compare it against Veins 5.1. As the impact of parallelization depends highly on the scenario configuration and the simulation hardware, we vary both in the case study.

For the simulation hardware, we consider three platforms, as detailed in Table 1: a mobile laptop, a typical desktop computer, and a more powerful workstation. Each hardware platform allows hyper-threading to virtually double the core count. We picked these as representatives for what researchers will typically have at their disposal when developing and researching wireless protocols. All code and configuration for the experiments is published as [5].

5.1 Simulation Scenarios

For the simulation scenarios, we picked two typical cases in VANET research as traffic scenarios: a densely populated motorway and an urban city. For both scenarios, the communication follows a static beaconing protocol, resembling an application like Cooperative Awareness (CA) messaging. The beacons are sent with a frequency

Table 1: Hardware platforms for the evaluation

platform	CPU	Cores	Threads
2-core i5 laptop	Intel i5-6200U	2	4
4-core i7 desktop	Intel i7-7700K	4	8
8-core R7 workstation	AMD r7-5800X	8	16

of 10 Hz and have a length of 350 B, as is typical for ITS G5 Cooperative Awareness Messages (CAMs) [21, 31]. The simulated transceivers have a sensitivity (`minPowerLevel`) and noise floor of −98 dBm [3]. The traffic simulated in Simulation of Urban Mobility (SUMO) is synchronized every 1 s and interpolated in between. All other settings follow the defaults of Veins 5.1. These settings yield plausible scenarios with well-populated wireless channels, showcasing the effects of interference on the simulation performance.

For the urban scenario, we used the Paderborn Scenario [6]. In it, more than 2300 vehicles driving through the city are simulated in the morning rush hour. The scenario is an example for a CA application in an urban area. Vehicles transmit with 200 mW or 23 dBm as is typical in such situations [16]. The most important factor in transmission success in urban environments is shadowing by buildings. Thus, the urban scenario uses Veins' `SimpleObstacleShadowing` model [27] in addition to the free space path loss model. With more than 50 000 buildings that have to be considered for each transmission, the `SimpleObstacleShadowing` model was found to have the biggest impact on the performance of this scenario. The scenario is simulated for 2.0 s in the benchmarks.

For the motorway scenario, we created a straight motorway with three lanes in both directions and a length of 5000 m. This scenario showcases applications with a densely populated channel and at high speeds. The motorway is filled to capacity with vehicles driving at a maximum speed of 130 km/h. In total, around 300 vehicles are driving at the same time. With this many other vehicles close by and no shadowing by buildings, transmit power is reduced to 20 mW (roughly 13 dBm). The most computationally complex model is the `VehicleObstacleShadowing` model [29] which runs in addition to the simple path loss model. It has an even higher impact than the building obstacle shadowing in the urban scenario and requires rolling updates of vehicle positions, which is also showcased with this scenario. The scenario is simulated for 10.0 s in the benchmarks.

5.2 Simulation Setup

The simulations runs in a steady state with a stable amount of vehicles and thus beacons over simulation time. To achieve this, the traffic scenarios are warmed up in SUMO before Veins starts simulating communication. For the urban scenario, the simulation state is loaded from a pre-saved state snapshot in SUMO. This reduces the time Veins has to wait for SUMO to warm up, which would otherwise skew timing measurements. Preliminary measurement similar to those for the real-time stability (see Section 6.3) proved that the speedup is stable across the simulation time. Simulations of shorter simulated durations yield similar speedup results and can be used to shorten the benchmarks.

For statistically stable results, each scenario was run 9 times on each platform: With 3 different seed sets in OMNeT++ (to gain independence from the PRNG sequences) and 3 times with exactly the same configuration (to gain independence of effects of the simulating platform). Measurements were taken on both scenarios and the 3 hardware platforms, leading to 54 simulation runs in total.

The results in the following sections have been obtained without runtime error or fingerprint checking and compiled in release mode, unless stated otherwise. This is to exclude further influence on the runtime performance. Verification runs have been performed before

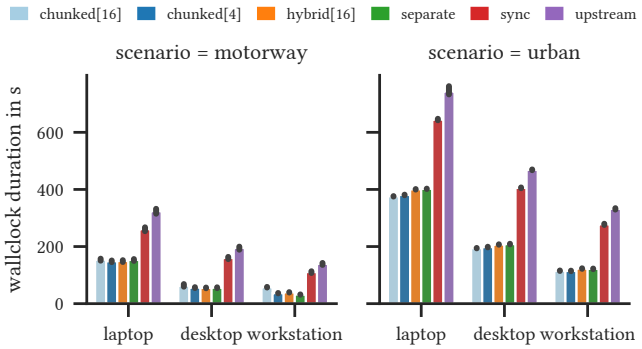

Figure 3: Simulation durations

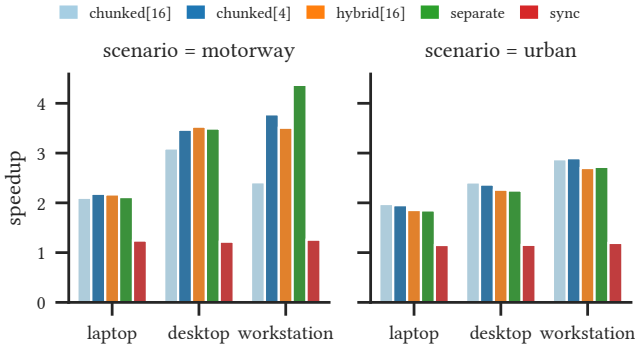

Figure 4: Simulation speedups

and not included in the measurements. Each hardware platform used background threads equal to the number of virtual (hyperthreaded) cores, i.e., twice the number of physical cores (c.f. Table 1).

6 RESULTS

The following sections show the results of the evaluation for the most relevant parallel strategy configurations to illustrate their performance. We present results for the *hybrid* strategy with 16 individually scheduled signals, and for the *chunked* strategy with a size of 4 and 16 signals per chunk. Under the name *upstream* we include results for the unmodified release of Veins 5.1 for comparison and as reference for speedup factor computation. Error bars in the plots indicate the 95 % confidence interval around the mean.

6.1 Speedup

The benchmark runs have shown that all parallel strategies can significantly shorten the simulation duration across all hardware platforms and in both scenarios. Figure 3 shows the mean durations for each scenario, platform, and parallel strategy, while Figure 4 shows the same data as speedups relative to *upstream*. Compared to the *upstream* version of Veins 5.1, all other strategies run faster, even the single-threaded *sync*. Thus, our (semantically equivalent) redesign of signal processing made the overall simulation faster by a factor of 1.15–1.25 even without parallelization. And aside all parallelization, the absolute numbers in Figure 3 also show the

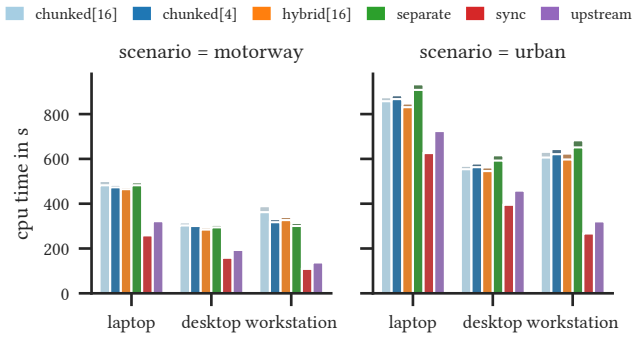

Figure 5: Total CPU (user) time spent during simulation

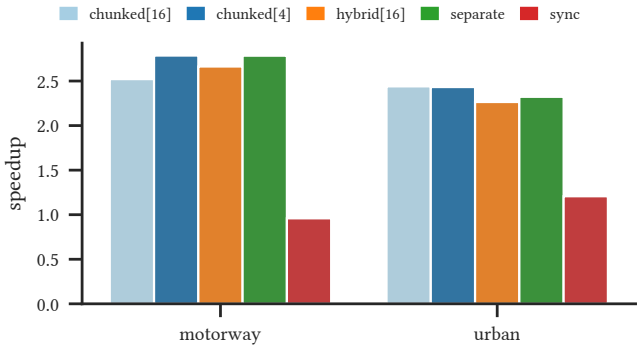

Figure 6: Simulation speedups for debug-builds on the 4-core i7 desktop platform

significant impact a more modern and powerful CPU can have on the simulation durations.

The difference in speedup between the two scenarios and the parallel strategies seem to increase with the computation power of the hardware platform: On the 2-core i5 laptop platform, the speedup for parallelized simulation lies at about 2, which matches it's physical core count, with very little difference between the parallel strategies and scenarios. On the 4-core i7 desktop platform, the parallel strategies show roughly similar speedups. But the obtainable speedup of the motorway scenario is much higher than the urban scenario. On one hand, this stems from the more computationally complex `VehicleObstacleShadowing` model in the motorway scenario. This also explains why the *chunked[16]* strategy obtains the lowest speedup: the main thread has to wait for the first large chunk to finish in the background, while other strategies can provide first results earlier. On the other hand, the higher total number of vehicles in the urban scenario increases the non-parallel portion of the code, e.g., updating vehicle positions or maintaining interferer frames, which reduces parallel speedup. The less complex building obstacle shadowing in the urban scenario also leads to the *chunked[16]* strategy obtaining the largest speedup on the 4-core i7 desktop platform. In this case the lower overhead of the larger chunks make it more efficient while even the large chunks are finished in time for the main thread. On the 8-core R7 workstation platform, the effects described for the 4-core i7 desktop platform are present in a more extreme way. For the motorway scenario, the *chunked[16]* strategy falls far behind while more fine-grained strategies like *chunked[4]* and especially *separate* yield the highest speedups by utilizing the powerful CPU and its many cores. On the contrary, for the urban scenario, the *chunked* strategy with its lower overhead again yields the highest speedups. The *hybrid* strategy typically lies in the middle ground, regardless of the platform or scenario, and yields a speedup of up to 3.5.

The amount of CPU seconds spent in user (brighter, bottom bars) and kernel/system (darker, top bars) mode shown in Figure 5 reinforce these findings. The *separate* strategy is very efficient on powerful platforms and complex models, but incurs more computation and synchronization overhead for less complex models. The *hybrid* strategy remains the most efficient on in all cases but the motorway scenario on the 8-core R7 workstation platform. But Figure 5 also shows the cost trying to exploit more parallelism on more powerful platforms: The amount of CPU seconds spent on single-threaded runs consistently gets smaller with more powerful platforms, indicating higher per-core throughput. But the total amount of CPU seconds spent on the 8-core R7 workstation platform is higher than on the 4-core i7 desktop platform with only half the core count. So the higher speedup is bought by a reduced efficiency. This probably results from the overhead in management and contention among the higher number of threads, which could be mitigated by manually reducing the number of threads for background processing in Veins. Though hardware techniques like frequency boosting may also benefit single-threaded code especially.

6.2 Debug Build Speedup

In addition to the optimized builds shown above, we also timed the debug builds of the two simulations. In contrast to the optimized builds, the code was not compiled with the `-O3` flag and contains debug symbols. For comparison, the mean simulation durations increase from 195 s to 1015 s and 469 s to 2399 s, for the motorway and urban scenario respectively. The resulting speedups on the 4-core i7 desktop platform are shown in Figure 6. In both scenarios, the parallelization can still improve the simulation duration by a factor of roughly 2.5. This is roughly similar to the speedup obtained with optimized builds (cf. Figure 4). So the parallelization may mitigate the extra cost of running in debug mode and help developers to get to erroneous sections of their simulations faster.

6.3 Real-Time Steadiness

For real-time applications, the simulation not only has to be fast enough, but also consistently fast enough in every step. Because even a single missed deadline can invalidate the whole run. To assess the steadiness of the parallelized code and the strategies in particular, we measured not only the total runtime but the speed of the simulation process over time. Specifically, this means the simulated seconds per wall clock second, which is also referred to as a real-time factor. OMNeT++ can output these values periodically, and we recorded them for multiple strategies in a modified version of the motorway scenario. The simulation time is increased to 600 s with a warm-up of 300 s. But only 45 % of the vehicles will be equipped with a network device (via Veins' `penetrationRate`).

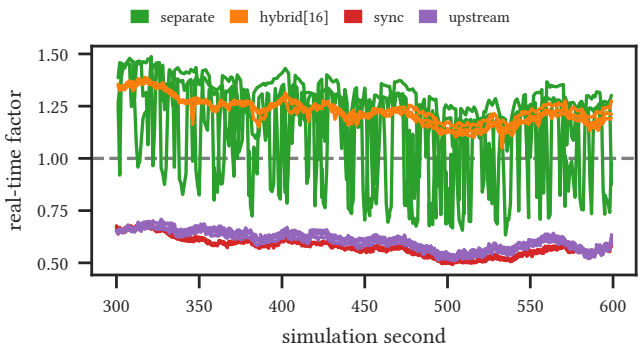

Figure 7: Real-time performance over simulation time on the 4-core i7 desktop platform

This reduces the computational complexity far enough that the 4-core i7 desktop platform can run the simulation in real-time.

Figure 7 shows the real-time factor for various parallel strategies across the simulation time on the 4-core i7 desktop platform. Lines of the same color represent different runs of the exact same configuration. The *chunked* strategies were omitted as they were out-performed by the other strategies for this scenario. Both the *separate* and *hybrid[16]* strategies are able to run the simulation time in less than that in wall clock time. But the *separate* strategy has a much more volatile profile and fell below the real-time factor of 1.0 multiple times. So the *hybrid[16]* strategy appears to be more suited for real-time applications.

6.4 Parallelization Effectiveness

To investigate the details of the performance of the simulation with asynchronous background processing, we sampled a profile for the *hybrid* strategy like we did for Veins in Section 4.1. The results of the influence of signal processing on the 4-core i7 desktop platform are shown in Table 2. For the motorway scenario, the number of samples for both the start and end events of signal processing have gone down by one to two orders of magnitude. The relative proportion of the samples of the main thread has gone down significantly, even though the wall clock time that the main thread ran also went down by a factor of ca. 3.5. The lower reduction at `processSignalEnd` is mostly due to the higher amount of non-parallelized functions called by `processSignalEnd` compared to `processNewSignal`. For the urban scenario, the same effect is visible for the start event of signal processing, while the reduction is much less pronounced for `processSignalEnd`. This is due to a much lower original influence of the attenuation model evaluation in the urban scenario, as interference and thus SINR-related late evaluation of attenuation models is much lower than on the motorway scenario.

6.5 Discussion

The best strategy depends on the scenario, hardware platform, settings, and requirements—e.g., real-time stability. The *hybrid* strategy is the versatile, being reasonably fast across all tested scenarios, efficient, and the most stable for real-time applications. But: other

Table 2: Percentage of main thread CPU profiling samples

Function	upstream	hybrid[16]
motorway scenario		
`processNewSignal`	3.0×10^{11} / 41.3 %	9.1×10^{9} / 8.5 %
`processSignalEnd`	3.1×10^{11} / 46.1 %	1.8×10^{10} / 17.1 %
urban scenario		
`processNewSignal`	1.3×10^{12} / 62.1 %	9.0×10^{10} / 14.2 %
`processSignalEnd`	1.7×10^{11} / 8.4 %	1.1×10^{11} / 17.4 %

strategies may yield better performance in some regard and depending on the platform. E.g., *separate* on powerful CPUs like on 8-core R7 workstation. We thus recommend to start out with *hybrid* and perform a few benchmarks for a given scenario to see if there are further speedups to be gained. As a general rule of thumb, stronger platforms and more complex tasks (e.g., attenuation models) benefit from more fine grained strategies, such as *separate*. On the other end of the spectrum, weaker platforms and less complex tasks benefit from strategies with less overhead, such as *chunked[16]*. Other factors to consider are the density of vehicles, transmit power, and maximum interference distance, which influence the number of received signal copies per sent message. The frequency of beacons or other messages and their respective length will have an impact not only on the amount of events themselves, but also on the amount of interference. Furthermore, the list of strategy types and configurations shown in this paper is not exhaustive. So custom strategies or parameters may provide better performance for other scenarios.

The amount of speedup to be gained through parallelization will always be limited by the portion of code that gets parallelized (*Amdahl's law*). For the scenarios shown in this paper, all signal attenuation computation has been moved to the background threads. The remaining runtime is mostly defined by the work that the main thread has to perform. This is mainly the overhead of the simulation kernel, e.g., maintaining and fetching from the FEL as well as distributing messages. But also portions of Veins still incur overhead, such as the data collection for attenuation models, storage of interferer frames, or vehicle mobility updates through TraCI. Finally, memory allocation and deallocation throughout these items may have significant impact on the performance, especially when it involves the managed object types of OMNeT++. This is already visible through the optimizations performed in the scope of this work: By removing the ubiquitous `Coord` class from the class hierarchy of OMNeT++ managed objects, significant overhead was removed when creating and freeing such objects. And by using shared data within of the `Spectrum` class, the amount of work needed when copying signals was lowered substantially. Further optimizing such portions of the code will thus not only reduce the overall computation time but also increase the effectiveness of parallelization, as the non-parallelizable portion of the code gets smaller.

When running a particular simulation study, there are a few other things to consider that impact the performance: Authors may want to consider reducing or disabling runtime verification—e.g., fingerprint checking in OMNeT++—or data output such as result collection or (event) logging. They (currently) have to run on the main thread, thus their impact on the performance of parallelized

simulations can be significant, especially if I/O-operations access slow mass storage. Finally, when running parameter studies or other inherently process-level parallel simulation suites, it is more efficient to disable parallelism and run each simulation in one thread (e.g., through the *sync* strategy).

7 CONCLUSION

In this work, we have shown the importance of fast simulation tools for wireless communication protocols and how asynchronous background processing can accelerate them in situations where traditional parallelization approaches struggle. Without changes to the outcome or the simulation model itself, this new concept can be applied to various simulation tools and target different portions of the code, running more efficient the more work can be offloaded to background tasks. We have shown this on the example of Veins and the evaluation of signal attenuation models. Across multiple hardware platforms and in two different scenarios, the implementation of asynchronous background processing has demonstrated reliable speedup of up to 3.5 (on a typical desktop platform). Speedup is also achieved in debug builds, helping developers reach relevant portions of their model faster by a factor of ca. 2.5. Furthermore, real-time applications, which can not utilize process level parallelism and need stable speeds across longer simulations, benefit greatly from the more that twofold continuous speedup.

In future work, the concept could be applied to other parts of the simulation or different simulation cores like ns-3. By combining the concept with other approaches of parallel and distributed simulation, even greater speedups may be possible, and the current bottleneck of the simulation kernel may be tackled. We aim to integrate our implementation into the next release of Veins so that the whole community can benefit from faster simulations.

ACKNOWLEDGMENTS

Research reported in this article was conducted in part in the context of the Hy-Nets4all project, supported by the European Regional Development Fund (ERDF).

REFERENCES

[1] Ismael Al-Shiab, Ayman Sabbah, Abdallah Jarwan, Omneya Issa, and Mohamed Ibnkahla. 2017. Simulating large-scale networks for public safety: Parallel and distributed solutions in NS-3. In *IEEE PIMRC 2017*. IEEE, Montreal, Canada. https://doi.org/10.1109/PIMRC.2017.8292761

[2] Peter D. Barnes, Matthew D. Bielejeski, David R. Jefferson, Steven G. Smith, David G. Wright, Lorenza Giupponi, Katerina Koutlia, and Colby Harper. 2019. S3: the Spectrum Sharing Simulator. In *2019 WNGW 2019*. ACM, Florence, Italy, 34–37. https://doi.org/10.1145/3337941.3337945

[3] Bastian Bloessl and Aisling O'Driscoll. 2019. A Case for Good Defaults: Pitfalls in VANET Physical Layer Simulations. In *IFIP WD 2019*. IEEE, Manchester, United Kingdom. https://doi.org/10.1109/WD.2019.8734227

[4] Fabian Bronner and Christoph Sommer. 2018. Efficient Multi-Channel Simulation of Wireless Communications. In *IEEE VNC 2018*. IEEE, Taipei, Taiwan. https://doi.org/10.1109/VNC.2018.8628350

[5] Dominik S. Buse. 2021. *Experiment Setup for "Accelerating the Simulation of Wireless Communication Protocols using Asynchronous Parallelism"*. Simulation Experiment Setup version 1.0. Zenodo. https://doi.org/10.5281/zenodo.5503502

[6] Dominik S. Buse. 2021. *Paderborn Traffic Scenario*. Traffic Simulation Scenario version 0.1. Zenodo. https://doi.org/10.5281/zenodo.4522058

[7] Dominik S. Buse and Falko Dressler. 2019. Towards Real-Time Interactive V2X Simulation. In *IEEE VNC 2019*. IEEE, Los Angeles, CA, 114–121. https://doi.org/10.1109/VNC48660.2019.9062812

[8] David Eckhoff, Alexander Brummer, and Christoph Sommer. 2016. On the Impact of Antenna Patterns on VANET Simulation. In *IEEE VNC 2016*. IEEE, Columbus, OH, 17–20. https://doi.org/10.1109/VNC.2016.7835925

[9] Richard M. Fujimoto. 2016. Research Challenges in Parallel and Distributed Simulation. *TOMACS* 26, 4 (May 2016). https://doi.org/10.1145/2866577

[10] Moritz Gütlein, Reinhard German, and Anatoli Djanatliev. 2019. Performance Gains in V2X Experiments Using Distributed Simulation in the Veins Framework. In *IEEE/ACM DS-RT 2019*. IEEE, Cosenza, Italy. https://doi.org/10.1109/DS-RT47707.2019.8958671

[11] Robert H. Halstead. 1985. MULTILISP: a language for concurrent symbolic computation. *TOPLAS* 7, 4 (Oct. 1985), 501–538. https://doi.org/10.1145/4472.4478

[12] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin Wong. 2019. Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++. In *IEEE IPDPS 2019*. IEEE, Rio de Janeiro, Brazil, 974–983. https://doi.org/10.1109/IPDPS.2019.00105

[13] R. Isermann, J. Schaffnit, and S. Sinsel. 1999. Hardware-in-the-loop simulation for the design and testing of engine-control systems. *Control Engineering Practice* 7, 5 (May 1999), 643–653. https://doi.org/10.1016/S0967-0661(98)00205-6

[14] David R. Jefferson. 1985. Virtual time. *TOPLAS* 7, 3 (July 1985), 404–425. https://doi.org/10.1145/3916.3988

[15] Stefan Joerer, Christoph Sommer, and Falko Dressler. 2012. Toward Reproducibility and Comparability of IVC Simulation Studies: A Literature Survey. *COMMAG* 50, 10 (Oct. 2012), 82–88. https://doi.org/10.1109/MCOM.2012.6316780

[16] Irfan Khan and Jérôme Härri. 2017. Can IEEE 802.11p and Wi-Fi coexist in the 5.9GHz ITS band ?. In *IEEE WoWMoM 2017*. IEEE, Macau SAR, China. https://doi.org/10.1109/WoWMoM.2017.7974358

[17] Georg Kunz, Olaf Landsiedel, Stefan Götz, Klaus Wehrle, James Gross, and Farshad Naghibi. 2010. Expanding the Event Horizon in Parallelized Network Simulations. In *IEEE MASCOTS 2010*. IEEE, Miami Beach, FL. https://doi.org/10.1109/MASCOTS.2010.26

[18] Georg Kunz, Mirko Stoffers, Olaf Landsiedel, Klaus Wehrle, and James Gross. 2016. Parallel Expanded Event Simulation of Tightly Coupled Systems. *TOMACS* 26, 2 (Jan. 2016). https://doi.org/10.1145/2832909

[19] Averill M. Law. 2007. *Simulation, Modeling and Analysis* (4 ed.). McGraw-Hill.

[20] Ioannis Mavromatis, Andrea Tassi, Robert J. Piechocki, and Andrew Nix. 2018. Poster: Parallel Implementation of the OMNeT++ INET Framework for V2X Communications. In *IEEE VNC 2018, Poster Session*. IEEE, Taipei, Taiwan. https://doi.org/10.1109/VNC.2018.8628429

[21] Rafael Molina-Masegosa, Miguel Sepulcre, Javier Gozalvez, Friedbert Berens, and Martinez Vincent. 2020. Empirical Models for the Realistic Generation of Cooperative Awareness Messages in Vehicular Networks. *TVT* 69, 5 (May 2020), 5713–5717. https://doi.org/10.1109/TVT.2020.2979232

[22] Christina Obermaier, Raphael Riebl, Christian Facchi, Ali H. Al-Bayatti, and Sarmadullah Khan. 2021. COSIDIA: An Approach for Real-Time Parallel Discrete Event Simulations Tailored for Wireless Networks. In *ACM SIGSIM PADS 2021*. ACM, Virtual, Online, 165–171. https://doi.org/10.1145/3437959.3459250

[23] Joshua Pelkey and George F. Riley. 2011. Distributed simulation with MPI in ns-3. In *ACM/ICST SIMUTools 2011*. ACM, Barcelona, Spain, 410–414.

[24] Patrick Peschlow, Andreas Voss, and Peter Martini. 2009. Good News for Parallel Wireless Network Simulations. In *ACM MSWiM 2009*. ACM, Tenerife, Spain, 134–142. https://doi.org/10.1145/1641804.1641828

[25] Ayman Sabbah, Abdallah Jarwan, Ismael Al-Shiab, Mohamed Ibnkahla, and Maoyu Wang. 2018. Emulation of Large-Scale LTE Networks in NS-3 and CORE: A Distributed Approach. In *IEEE Milcom 2018*. IEEE, Los Angeles, CA, 493–498. https://doi.org/10.1109/MILCOM.2018.8599762

[26] Y. A. Sekercioglu, A. Varga, and G. K. Egan. 2003. Parallel simulation made easy with OMNeT++. In *European ESS 2003*. Delft, Netherlands.

[27] Christoph Sommer, David Eckhoff, Reinhard German, and Falko Dressler. 2011. A Computationally Inexpensive Empirical Model of IEEE 802.11p Radio Shadowing in Urban Environments. In *IEEE/IFIP WONS 2011*. IEEE, Bardonecchia, Italy, 84–90. https://doi.org/10.1109/WONS.2011.5720204

[28] Christoph Sommer, Reinhard German, and Falko Dressler. 2011. Bidirectionally Coupled Network and Road Traffic Simulation for Improved IVC Analysis. *TMC* 10, 1 (Jan. 2011), 3–15. https://doi.org/10.1109/TMC.2010.133

[29] Christoph Sommer, Stefan Joerer, Michele Segata, Ozan K. Tonguz, Renato Lo Cigno, and Falko Dressler. 2015. How Shadowing Hurts Vehicular Communications and How Dynamic Beaconing Can Help. *TMC* 14, 7 (July 2015), 1411–1421. https://doi.org/10.1109/TMC.2014.2362752

[30] Lukas Stratmann, Dominik S. Buse, Julian Heinovski, Florian Klingler, Christoph Sommer, Jan Tünnermann, Ingrid Scharlau, and Falko Dressler. 2019. Psychological Feasibility of a Virtual Cycling Environment for Human-in-the-Loop Experiments. In *Jahrestagung INFORMATIK 2019, ICT4VRU Workshop*, Claude Draude, Martin Lange, and Bernhard Sick (Eds.), Vol. LNI P-295. GI, Kassel, Germany, 185–194. https://doi.org/10.18420/inf2019_ws21

[31] Martinez Vincent and Friedbert Berens. 2018. *Survey on ITS-G5 CAM statistics*. TR 2052, V1.0.1. C2C-CC. https://www.car-2-car.org/fileadmin/documents/General_Documents/C2CCC_TR_2052_Survey_on_CAM_statistics.pdf

[32] Xiang Zeng, Rajive Bagrodia, and Mario Gerla. 1998. GloMoSim: a Library for Parallel Simulation of Large-scale Wireless Networks. In *Workshop PADS 1998*. IEEE, Banff, Canada, 154–161. https://doi.org/10.1109/PADS.1998.685281