Asynchronous Background Processing for Accelerated Simulation of Wireless Communication on Multi-Core Systems

Dominik S. Buse[a,b], Georg Echterling[a], Falko Dressler[b]

[a]*Department of Computer Science, Paderborn University, Germany*
[b]*School of Electrical Engineering and Computer Science, TU Berlin, Germany*

Abstract

Discrete event simulation (DES) is an important tool for the development and analysis of wireless networks. However, with increasing network size and complexity, the computational effort and simulation time increases significantly, often exponentially. This increase in response time may be critical if DES is interfacing real-time systems like Hardware in the Loop (HIL) or network emulation. It also slows down development cycles of users designing or debugging simulation models. Most popular DES software packages run single-threaded. Thus, they achieve only limited performance improvements from more modern multi-core CPUs. At the same time, existing approaches for parallel simulation of networks do not perform well on wireless systems or require complex paradigm shifts in simulation models. In this paper, we propose Asynchronous Background Processing (ABP) to accelerate the simulation of wireless communication on multi-core systems. By moving expensive computation from the main thread into asynchronous tasks computed by background threads, it accelerates the progression of events and thus reduces response time. Tasks are started as early as possible to exploit the time the main thread spends processing other events, ideally providing results before they are needed in the simulation. We showcase the application of ABP using Veins, a popular vehicular network simulator, demonstrating speedups of up to 3.5 on typical desktop platforms. We further perform an in-depth analysis using advanced profiling techniques to investigate the effectiveness of the parallelization and guide further optimizations.

Keywords: Parallel simulation, wireless network simulation, asynchronous parallelization, vehicular networking

1. Introduction

Wireless systems have become ubiquitous in today's world: Smartphones, Edge Clouds, Connected Cars, the Internet of Things. The applications they run and the protocols they rely on became ever more complex, leading to ever more complex systems. Designing these systems and investigating the various effects within them thus also becomes ever more complicated [1]. To tame this complexity, developers and researchers turn to simulation software. It allows running experiments reliably and reproducibly for setups that would cost fortunes to set up in real hardware, while also allowing to inspect and record minute details from any involved device. So, it is no wonder that network simulators, typically using discrete event simulation (DES), have become such a common tool [2]. However, many common DES tools, like OMNeT++ or ns-3 have a problem with the increasing scale of networks: Their implementation is predominantly single-threaded [3]. Thus, the major gains in processor performance of the last decade that stem from increasing CPU core count, scarcely help to speed up network simulation.

This may not be a problem for large-scale parameter studies of wireless simulations. Process-level parallelism is easy to achieve if there is simply a large collection of independent simulation configurations to run. However, relying on this is not always possible—often it is response time (or delay), rather than throughput, that needs to be optimized. On the one hand, large-scale simulation studies typically lie at the end of a development or research process. Before that, the model must be carefully designed, implemented, debugged, and adapted [2]. This loop typically only needs one instance of a simulation and it needs it to reach a certain point at which a bug may occur, repeatedly. To make matters worse, this often involves running non-optimized debug builds of code. On the other hand, there are scenarios in which even for finished simulation models, there is only one instance of interest. This is often the case when the simulation is coupled to some other system, especially when real-time elements are involved (e.g., Hardware in the Loop (HIL) setups, human-facing systems, on-policy training of agents via reinforcement learning) [4, 5].

This is particularly relevant for the simulation of wireless communication [6]. Approaches known from other domains of simulation—even wired networking simulation—often do not work. On the shared medium of the wireless channel, lookahead times are low and may become even lower with an increase in complexity, i.e., number of

Email addresses: `buse@ccs-labs.org` (Dominik S. Buse), `georg@echterling.net` (Georg Echterling), `dressler@ccs-labs.org` (Falko Dressler)

nodes, complicating the application of Logical Processes (LPs) [7]. Optimistic synchronization, like the *Time Warp* algorithm [8], would see a large number of rollbacks due to high interference in dense scenarios. Also, multi-threaded tools that have been proposed so far lack widespread adoption. They either require a re-design of the simulation framework and simulation models alike [9]. Or they sacrifice reproducibility (i.e., determinism) or accuracy for the sake of performance, often in the context of interference modeling [10]. For all these reasons, popular simulation frameworks like ns-3 or OMNeT++/INET have not yet adopted a multi-threaded model for wireless simulations. Overall, it is still hard to create general forms of parallelized architectures for wireless system simulation.

In this paper, building on our previous work in [11], we present the concept of Asynchronous Background Processing (ABP) to achieve accessible multi-core performance for the simulation of wireless communication. It relies on isolating expensive computation out of the main thread to achieve speedup without changing the core of the simulation. By exploiting semantic relationships between events, which are common in DES, background computation tasks are started as early as possible, such that their results are ideally already available before they are needed by the main thread of the simulation. This way, the main thread may advance simulation time faster, resulting in overall simulation speedup. All while the processing of simulation events is untouched, making ABP virtually transparent to the user.

As an application of ABP, we present a comprehensive case study on Veins [12], a widely adopted simulator for vehicular networks based on OMNeT++. By parallelizing the signal attenuation models in Veins, we showcase the speedups achievable to a wide variety of simulations, as the evaluation of these models dominate the performance of many larger simulation scenarios. At the same time, this acceleration will benefit many users with minimal impact to their models, e.g., application layer protocols. Along the case study, we guide through the entire process of applying ABP and holistically examine the outcome of the parallelization.

Our main contribution can be summarized as follows:

- We present Asynchronous Background Processing (ABP), a concept to speed up wireless simulations without deep changes to the simulation model;
- we perform a case study of ABP on Veins, a commonly used vehicular network simulation software; and
- we study the process of ABP application, from initial potential estimation, to implementation, result verification, and in-depth performance evaluations.

2. Related Work

The typical tools 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 LPs and conservative synchronization with lookahead times. E.g., OMNeT++ has supported the null message algorithm for a long time [13]. Its implementation uses the Message Passing Interface (MPI) and utilizes link delays as lookahead. Similar approaches have been implemented for ns-3 [14], 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" [14].

Pioneering work for parallel simulation of wireless systems has been done in [15]. 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, and more 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 Vehicular Ad-hoc Networks (VANETs) on a motorway.

A recent study [16] surveyed approaches for ns-3 for the simulation of 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 multi-threading 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 [17]. 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 must 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* [18, 7] 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 intervals of time, by adding a duration. Events may only change the simulation state after their duration has passed. Thus, events with overlapping time intervals cannot depend upon each other—they either would not have been allowed to start or would not be able to use each other's results. Thus, overlapping events can be scheduled to run in parallel. This even works for wireless mobile simulation and the authors evaluate *HORIZON* for an LTE network. However, it requires significant changes to the simulation kernel and assignment of a duration to a sizable 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 [19]. 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]. It showcases the possibility of a distributed simulation of a Green Light Optimal Speed Advisory (GLOSA) application. 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) [20]. 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 [21] 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 light-weight fibers, which implement LPs, for parallel execution. *COSIDIA* promises "fully deterministic functional behavior 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 number of nodes 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 nodes 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 extremely 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 since the introduction of antenna models [22] in Veins 4.5, the distance must be configured by the simulation author. Tuning this parameter can have a significant impact on the performance but also the result of the simulation and provides a trade-off between accuracy and speed. Choosing the right value may require extensive experience and preliminary studies though, as it depends on many factors such as channel models, transmission technology, shadowing, and more.

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* [23]. 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. Thresholding requires that further models can only decrease the received signal strength. Attenuation models that may increase the received signal strength (e.g., two-ray interference) must 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 vastly different speedups depending on the channel models involved and only speeds up this part of the simulation. But in practice it can provide useful speedup and is compatible with approaches to speed up simulation using parallel processing.

3. Asynchronous Background Processing (ABP)

Asynchronous Background Processing (ABP) is a concept to speed up simulations of DES software. It proposes moving computationally expensive tasks from the main thread to background worker threads. These computation tasks are started as early as possible in simulation time and are concurrent to the main thread. Previously synchronous (i.e., blocking) computations thus become asynchronous, which introduces the potential for parallel execution. This allows the main thread to advance simulation time faster, resulting in a speedup of the overall simulation. In contrast to other parallelization techniques for DES, ABP only moves *computation tasks* out of the main thread—not events, entire simulation modules, or other entities of the simulation itself. This makes ABP much easier and less invasive to apply to existing simulation software. This section explains the details of ABP, how it can be applied to DES, and the conditions that have to be met for it to work.

3.1. Time Progression in Sequential DES

Typical DES tools used for wireless systems, 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.

There are two important notions of time when discussing simulation: Simulation time, i.e., the time within the simulation, and wallclock, i.e., time physically passing while a computer is running the simulation software (cf. [24]). Typically, there is no link between simulation time and wallclock time (except that both never move backwards). The progression of simulation time may vary drastically: Multiple close-by events with high computational complexity may take a long wallclock time to compute but only advance the simulation time a little. Meanwhile, large gaps between events can lead to fast progression of simulation time without much computation time spent.

As an example, consider the simulation of a wireless network consisting of a few close-by nodes that each broadcast messages every second or so (cf. Figure 1). Each broadcast will lead to a burst of close-by events in all receiving nodes with very little time in between them—in the order of nanoseconds—due to propagation delay. However, the time between two broadcasts may be orders of magnitude larger, depending on the configuration and number of nodes, e.g., in the order of milliseconds.

How long it takes to compute an event depends on the complexity of the code executed within that event. And this computational complexity may vary greatly as well: Some events only save some value or schedule another event for later, while others execute complex computations of models or system behavior. As a result, the code within individual events may take much more time to compute than the simulation kernel working through the FEL itself.

Individual events usually represent a discrete point in time for isolated entities in the simulation. There is often a semantic relationship between two events. E.g., the event in which the broadcast is transmitted by the sender and the reception event of another node (trigger and reaction). Or the reception events of the same broadcast on two different

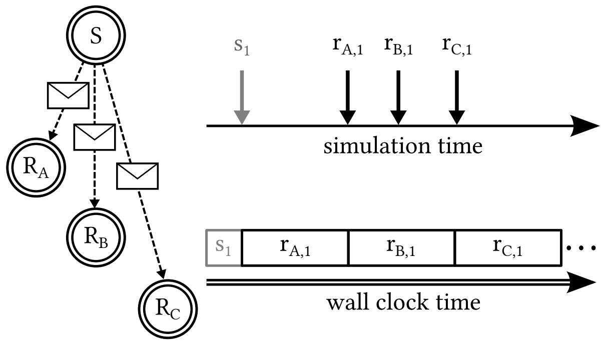


Figure 1: Example simulation of a broadcast in a wireless network.

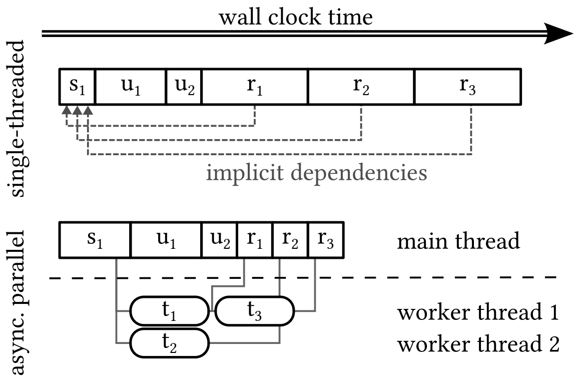


Figure 2: Time progression in single-threaded (top) and asynchronous parallel (bottom) simulations.

nodes (two reactions to the same trigger). However, these semantic relationships are typically not encoded into the simulation directly. Instead, they are 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.

3.2. Extracting Computation Tasks

The way DES work can be exploited to speed up the simulation through ABP: By offloading expensive computations between two events from the simulation main thread to some background (worker) thread. Assume again the example from the previous section, with one node sending a message to three receivers. This could result in a FEL and computation as illustrated in Figure 2. It consist at least of the sending event s_1, and three receiving events r_1, r_2, and r_3 at receivers 1, 2, and 3, respectively. Naturally, the sending event s_1 takes place before the receiving events, which in turn are sorted by distance (due to the propagation delay). But there are also some unrelated events u_1 and u_2 between sending and the first reception, e.g., modeling post-transmission code running on the sender. In this example model, reception events are computationally expensive, as they contain some channel or error models to determine the success of the transmission. These computations could also be evaluated as part of the sending event s_1, as all information needed for them is known at that point.

Though that would not change the total computation time significantly.

With ABP however, the computations can also be extracted into computation tasks and run in background worker threads. The bottom half of Figure 2 illustrates this for a scenario with two worker threads: During s_1, the computation tasks are created by collecting relevant information and scheduling the tasks t_1, t_2, and t_3, resulting in a slightly more complex event s_1. The worker threads then start processing the tasks, while the main thread continues with the unrelated events u_1 and u_2. By the time the main thread starts processing r_1, the tasks t_1 is already done, its result can be obtained, and r_1 can be completed much faster. And even if it was not ready at that time—resulting in a stall of the main thread waiting for the result of t_1—r_1 would still have concluded earlier as part of its computation was started earlier as well. The same is true for r_2 and t_2, as the two worker threads processed t_1 and t_2 in parallel. And even though t_3 only gets processed once t_1 is done, its result is ready before the main tread arrives at r_3. This leads to a significant reduction of the wallclock time needed to simulate the transmission through the utilization of multiple cores.

Naturally, there can be more than one background task active at the same time (both simulation and wallclock time). Though the actual number of tasks 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 computation tasks. Having many background tasks scheduled at the same time makes the procedure even more efficient, 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 ABP has similarities to other offloading techniques, e.g., to GPU offloading, it exploits the nature of DES by advancing the simulation while the offloaded tasks are being processed. This may yield greater speedups than just using parallel algorithms for the computationally expensive operations (see Section 7). Also, the implementation of the model and the simulation core itself did not have to be modified or re-implemented, like it would be necessary 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 background tasks and their synchronization between threads is using Futures and Promises [25]. When the task is offloaded, the main thread receives a Future instead of the actual result. 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. Though this is typically abstracted away by the asynchronous offloading library. Together, the Future and the Promise form a channel between the main thread and the offloaded task, which takes care of all further synchronization. The main thread can attach the Future to the event or simulation object that will eventually request the result of the computation.

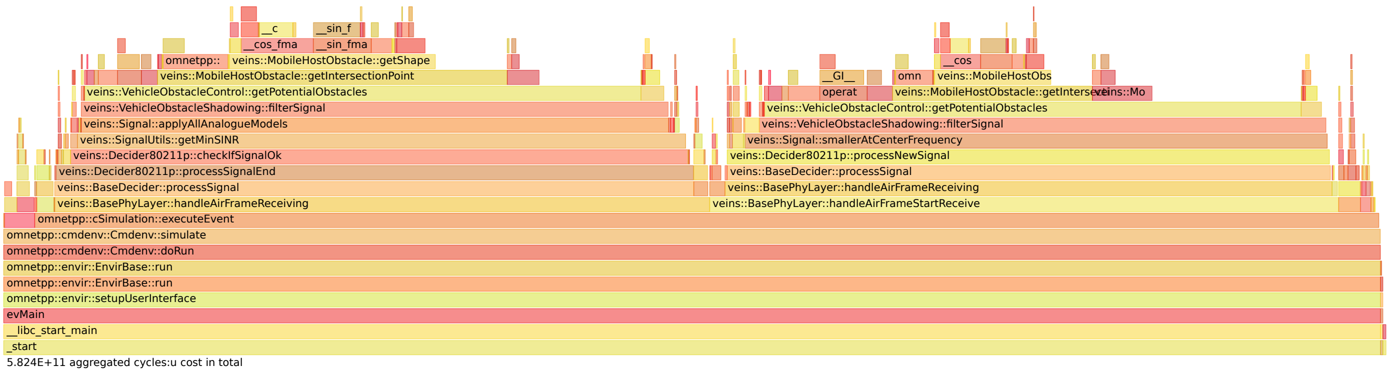
3.3. Challenges

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 Pseudorandom Number Generator (PRNG): There may be multiple background tasks being processed at the same time in different threads, so the order of the calls to a PRNG becomes non-deterministic. This could lead to non-reproducible changes in the simulation outcome, which must 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.

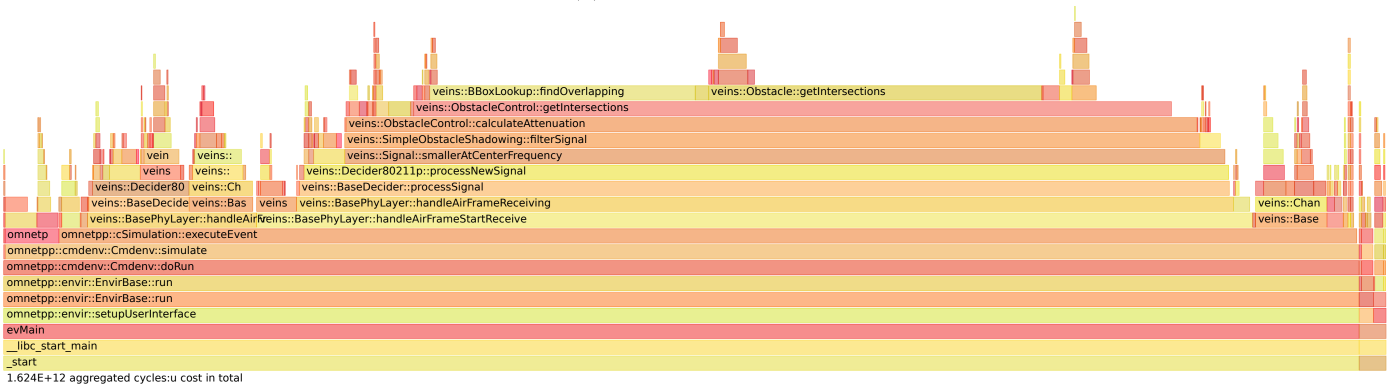
In practice, this often means moving the start of the computation to the point in simulation time at which they first become possible. It may even be useful to include processing delays of the system represented by the model to create more opportunities for the completion of background tasks while the 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, e.g., sorting or numerical optimization.

4. Application Case Study: Veins

We demonstrate our Asynchronous Background Processing (ABP) concept by applying it to Veins [12], a state-of-the-art VANET simulator. This section describes the process we followed to serve as a case study for the application of ABP. First, we examine the performance profile of Veins 5.1 (the most recent release when we started the application of ABP) to discover potentials for parallelization. This includes an analysis of the computationally expensive portion of the code we found for applicability of our concept. Based on that, we implement ABP in Veins and explain the details of the challenges encountered during the implementation. Finally, we describe how we ensured the equality of the results of the original sequential version of Veins and our parallelized implementation.



(a) Motorway scenario.



(b) Urban scenario.

Figure 3: Flame graphs of performance profiles of Veins 5.1 (recorded on the 8-core workstation platform.)

4.1. Finding Potentials

Finding portions of the code to be turned into background tasks is possibly the hardest part of applying ABP. The same real-world process can be simulated with vastly different implementations, with highly different potentials for parallelization, yet still producing the same results. By understanding what actual results and behavior the simulation should produce, the implementation may be changed to be more suited for ABP. The important part to remember is that computations for background tasks have to be isolatable from the simulation kernel. This may not be immediately possible if the original implementation was not designed with this principle in mind. A complete re-modelling and re-implementation is often not possible due to the required time and effort. But in many cases re-thinking how a small portion or aspect of the simulation is modelled and implemented may have great impact on the parallelization potential. So, it makes sense to always keep in mind what is the necessary behavior to be simulated and what is just an implementation detail.

Performance analysis tools can be of great help to find potentials in a concrete implementation. One of the most common tools for this is `perf`, a sampling profiler for Linux. While the simulation runs, `perf` records a *performance profile* by sampling the currently active stack frame of a program at regular intervals, e.g., 10 ms. In comparison to many other profiling approaches, sampling is very efficient and simple to apply. The program under test does not have to be modified in any way, neither by the programmer nor through another program or compiler. And the process of sampling has a low overhead, typically only introducing a few percent of increased program runtime. Thus, the program executes very similar under profiling to how it would without. As a downside, it can only indirectly tell how much of an impact a certain function has on the overall performance: It only statistically tells how many of the samples of the performance profile were found to involve a certain function, but not how long a single call of said function took or how often it was called. However, for finding computationally intensive parts of the program, this is typically sufficient. Functions with a high proportion may be candidates for background tasks.

To find potentials in Veins by identifying computationally complex code, we profiled two typical simulation scenarios using `perf`. The scenarios are described in detail in Section 5.1 and will later be used for the evaluation of our parallelized version of Veins. We plotted the performance profiles as *flame graphs*, which are shown in Figure 3. Both have been filtered to include only samples from the main thread on the OMNeT++ simulation, e.g., excluding samples of the Simulation of Urban Mobility (SUMO) process. Flame graphs (see [26]) can visualize the stack samples collected in the performance profile from bottom (program entry point) to top (most deeply nested stack frame). Each box represents a certain function, and other functions called are stacked on top. The width of each box is proportional to the relative number of samples of the function at this level of the stack frame. This provides an (estimated) overview of which functions of the code base consume the most computation time.

In both flame graphs, on the very bottom, we see the overall simulation process, managed by the OMNeT++ simulation kernel. The main loop of the simulation is located in `omnetpp::cmdenv::Cmdenv::simulate()`, on the eighth level from the bottom. Only a tiny fraction of sample lies to the right, outside of its stack, which are mostly about simulation startup and memory allocation, and can be neglected for our analysis. The layer immediately above is dominated by two stacks: The stack on the left (without a name in the figure) contains the code to maintain the FEL, i.e., adding, sorting, and fetching events. As this is part of the simulation kernel itself (and a very small portion of the samples, too), we do not consider it for parallelization. On the right there is `omnetpp::cSimulation::executeEvent`, which runs the code within each event of the FEL, and makes up the vast majority of samples. One level further above, we can see the two types of events that make up the majority of the computation time and thus give us the first important hint on what to parallelize. `handleAirFrameReceiving` (47.3 % and 12.3 % of the samples for motorway and urban scenario, respectively) and `handleAirFrameStartReceiving` (45.5 % and 73.3 %) of the `veins::BasePhyLayer` module. This is where the code of the Veins simulation begins in the call stack (compared to the OMNeT++ simulation kernel below). It is also where this simulation spends most of its wallclock time. Other modules like the Medium Access Control (MAC) layer or interaction with SUMO via Traffic Control Interface (TraCI) are negligible in caparison, so we rule them out for now for parallelization efforts.

Moving up the stacks, we see what code in particular is responsible for the computation load. Both stacks prominently contain the `filterSignal`, a method of a class derived from `AnalogueModel`. For the motorway scenario (see Figure 3a), it is the `VehicleObstacleShadowing` analogue signal attenuation model. In the urban scenario (see Figure 3b) it is the `SimpleObstacleShadowing`, with the same structure, but a much stronger influence of the method `handleAirFrameStartReceive`.

Our knowledge of the codebase also tells us that these attenuation models are a somewhat encapsulated portion of the code with a clear interface: Different subclasses of the `AnalogueModel` class implement the virtual `filterSignal` method, which takes a `Signal` object and attenuates it according to the specific model. This is a good fit for what we are looking for. These functions are relatively pure computations of an isolated model within the simulation with high computational complexity and a defined interface. So we will try to draw the line there and use this interface to separate the main simulation thread from tasks for parallel processing. Furthermore, note that both stack frames are concerned with the *receiving* side of a transmission, meaning there must have been a transmission event before. This might be exploited to schedule background computation in advance, ideally producing computation results even before the main thread reaches the reception events.

If the outcomes would not have been as favorable at this point, we could have moved further up the call stacks in search for parallelization potential. E.g., we can see that most work within the `VehicleObstacleShadowing` module is spent finding potential obstacles and computing intersection points and the `SimpleObstacleShadowing` module is dominated by similar functions, but from different helper classes. So, these could have been other options for parallelization, though tied to a single specific attenuation model, a less general interface and semantic relation to previous events.

4.2. Signal Processing in Veins

We previously (see Section 4.1) identified that the biggest potential for parallelization for Veins lies in the computation of wireless signal transmission and attenuation. To understand how we modified Veins for ABP, we first describe how it worked before we applied the new concept.

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 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.

The `ChannelAccess` module is the superclass of the PHY layer hierarchy and takes care of disseminating messages on the wireless channel. Whenever such a wireless message, named `AirFrame` in Veins, is sent, `ChannelAccess` creates a copy of this `AirFrame` for each potential receiver and sends that `AirFrame` copy to it. Potential receivers are all other NICs within the *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 over the wireless channel, e.g., timing, frequencies, and power levels [23]. 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 a time offset for the propagation delay between sender and receiver (cf. Figure 4).

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 contains the evaluation of the attenuation models of the `Signal`

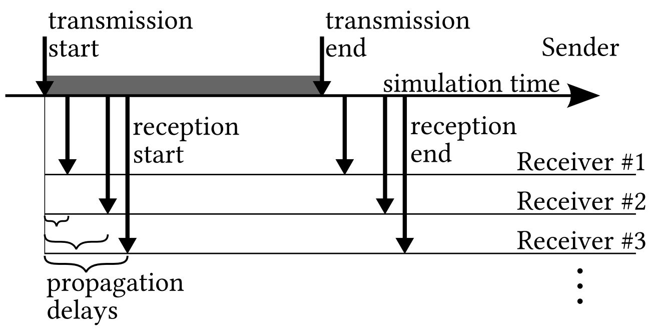


Figure 4: Timeline of the transmission of an `AirFrame` in Veins.

to compute the received power levels (including antenna gains [22]). If *thresholding* [23] 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 Background Processing in Veins

To implement ABP for the acceleration of signal attenuation models in Veins, we had to overcome a several challenges, which are discussed in this section. First, we had to isolate the code that runs the computation of attenuation models from any interaction with the simulation kernel. This required the code to collect and store all data needed for said computation in advance, before background tasks are spawned. Then, we needed to move the start of the computation from the reception events to the sending event. Only at this point, we were able to push the computation into background tasks and set up asynchronous result retrieval, which required the integration of an asynchronous task library. As spawning too many tasks can lead to significant overhead, we had to include task bundling strategies for cross-platform efficiency. And to maintain efficient execution when multi-threading is disabled, we had to re-integrate Veins' thresholding mechanism.

We started by gathering all received signal copies into a `SignalGroup` object within the `ChannelAccess` module. Whenever an `AirFrame` is sent, a `SignalGroup` is created. The `SignalGroup` contains all copies of the signal that are sent out to potential receivers within maximum interference distance, and manages the computation of their attenuation models. It further 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. Finally, it also efficiently stores the individual data per receiver, e.g., the receiver information, propagation delays, attenuated power levels, and applied attenuation models. All this storage is isolated from the rest of the simulation model and kernel in order to avoid conflicting access from the main thread and worker threads. 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.

Upon the construction of the `SignalGroup`, instances of each relevant signal attenuation model class are instantiated as well. Through polymorphism, each attenuation model can implement data collection for all special information needed for its computation. This way, for example, the `VehicleObstacleShadowing` model can obtain a snapshot of the positions of all vehicles at the time of signal sending. More generally, models requiring random numbers can draw them from PRNGs in advance and store them for later use during computation. All these data collection steps are performed in the main thread, making them deterministic and avoiding data races. The code to compute the results of the signal attenuation models (that will be run in the background threads) also was adapted to only use the safely stored values, isolating it from the running simulation model of the main thread.

Once a `SignalGroup` of fully constructed, it is scheduled for background computation. The main thread completes the remainder of the sending event in the `ChannelAccess` module, i.e., by distributing the `AirFrame` copies to the potential receivers and informing upper layers. It then continues with the next event in the FEL, while background threads perform the computationally expensive signal attenuation. 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. 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. The resulting (partially) attenuated signal is then stored in SignalGroup and a Promise object is fulfilled to indicate completion. A receiver can then safely access it from the main thread through the connected Future object attached to the SignalInstance. If the computation is not yet complete by that point in time, the access through the Future object will automatically block until the background task is done.

To efficiently implement background tasks, we used the *TaskFlow* library [27] 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 discard it due to its low and unpredictable performance.

We further introduced the ParallelStrategy abstraction, which decides how received signal copies are bundled and made it runtime configurable. Each SignalGroup contains multiple (often tens or even hundreds) of received signal copies, each with a set of attenuation models to evaluate. 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. This may work well on powerful platforms with high core counts, but diminish speedup on less powerful platforms. So, through a ParallelStrategy, received signal copies can be grouped and scheduled as a chunk. It schedules bundles of received signal copies to be computed on the background threads and provides a Future-like interface to await and retrieve the results for each SignalInstance. This reduces overhead but could in turn 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, the ParallelStrategy can be selected at runtime, when starting the simulation.

We 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.

With these changes, we were able to solve all challenges discussed above and implement ABP in Veins. While they are specific to the signal attenuation in Veins, we expect that applying ABP to other software will likely turn up similar issues, e.g., when balancing the amount of parallelism with the overhead incurred by it. So we hope these deliberations may guide the reader when parallelizing their own software.

Some further changes were made to Veins, which also improved performance and reduced 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

Extending a simulation implementation to utilize multiple cores should not change the results of the simulation. Ideally, there should be no deviation in any observable behavior and output of the simulation at all. Built-in mechanisms of simulation frameworks like result hashes (e.g., *simulation fingerprints* of OMNeT++) can often easily verify that two implementations yield identical results. This could proactively avoid many counter-arguments to using the parallel implementation and make the single-threaded and parallel implementation interchangeable.

But not allowing any deviation may leave potential speedup on the table. Because this often means staying true not only to the model of the system to be simulated, but also to the implementation details of this specific codebase. In addition, virtually all network simulation models contain stochastic elements, like numbers drawn from PRNGs. Simulations are often run with different seeds for statistical stability, resulting in distributions of results rather than exact values. If a parallel implementation of the simulation produces results that do not change the distribution of results, it can be considered equivalent in a statistical sense. Or differently put: statistically indistinguishable from the single-threaded implementation.

The optimization potentials opened up by this looser definition of equivalence are manifold: The order in which numbers are drawn from PRNGs may be varied, which is essential to gathering them before scheduling background computation tasks. Data from the running simulation, such as positions of mobile hosts, could be collected at slightly different points in simulation time. This allows moving code to another event to schedule computation in advance, while still producing results that are practically the same yet numerically different than those of a single-threaded implementation. And sometimes even the order of execution of small portions of code could impact the numeric results, e.g., rounding errors of floating point values, which are no longer an issue with this verification approach. Note that these examples only focus on computation and data collection, but do not introduce new events or messages from the perspective of the simulation kernel. They thus only affect the *implementation* of the simulation, not the model of the system being simulated. While changing the model as well is a possible way for even more aggressive parallelization and optimization, it requires even more careful verification of the resulting behavior of the simulation.

For practical purposes, we recommend a two-step process: First, make all the changes that will require stochastic equivalence checking. E.g., moving simulation data and PRNG number collection to different events. This may require careful planning, but once it is done, only one thorough verification of stochastic equivalence needs to be performed. Once the results are verified, generate a set of result hashes for this new version of the implementation. Then, as the second step, start implementing ABP. This can now be done in an iterative process, as the verification can easily be checked via the result hashes. If necessary, one could still go back to step one and repeat the process. But keeping the number of thorough verification passes low should make the development process significantly faster and more reliable.

Moving the point in simulation time at which data for the evaluation of signal attenuation model are collected introduced some small deviations in the results. For most vehicles and transmissions, there was no change at all. But in some cases, 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). This in turn leads to different outcomes of the signal attenuation in the `VehicleObstacleShadowing` model. E.g., if a vehicle is now in the line of sight of a transmission which reduces the received power levels of the signal. And this in some cases leads to further propagation of changes, as the different received power levels may lead to more or less random numbers being drawn in the bit error model. These effects and the resulting point of diversion in the results could be found in the trace logs of the simulation as expected.

To ensure these effects would not change the statistical outcome of the simulation, we performed a large set of replications with different seeds for both the original Veins 5.1-equivalent code and our parallelized implementation. We recorded the attempted and successful transmissions for all vehicles as well as the transmission start time. With that, we produced the distribution of attempted and successful transmission per vehicle per beacon interval (100 ms). These distributions did not significantly differ for the two different versions of the code. Note that the result deviations only occurred when comparing Veins 5.1 with our parallelized implementation. The modified code always behaves deterministically, regardless of the chosen hardware platform, number of threads, or `ParallelStrategy`.

Most of the other changes to Veins described in the previous sections, 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++ as the second step of the two-step process described above.

5. Experiment Setup

To showcase the speedup possible in Veins with ABP 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 of them.

For the simulation hardware, we consider four platforms, as detailed in Table 1: a mobile laptop, a typical desktop computer, a more powerful workstation, and a server with two CPU sockets. The first three hardware platform allows hyper-threading to virtually double the core count, while it was disabled for the 16-core xeon server platform. 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 [28].

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 of 10 Hz and have a length of 350 B, as is typical for ITS G5 Cooperative Awareness Messages (CAMs) [29, 30]. The simulated transceivers have a sensitivity (`minPowerLevel`) and noise floor of −98 dBm [31]. The traffic simulated in 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.

platform	CPU	Cores / Threads
2-core laptop	Intel i5-6200U	2 / 4
4-core desktop	Intel i7-7700K	4 / 8
8-core workstation	AMD r7-5800X	8 / 16
16-core xeon server	2xIntel E5-2670	16 / 16

Table 1: Hardware platforms for the evaluation.

We based the urban scenario on the Paderborn Scenario (see [11]). 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 [32]. The most important factor in transmission success in urban environments is shadowing by buildings. Thus, the urban scenario uses Veins' `SimpleObstacleShadowing` model [33] 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 [34] 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 simulation runs in a steady state with a stable number of vehicles and thus beacons over simulation time. This makes measured speedups mostly independent of the point in time within the simulation. 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 needs 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 and are not included in the measurements. Each hardware platform used background threads equal to the number of virtual (hyper-threaded) cores available (cf. Table 1).

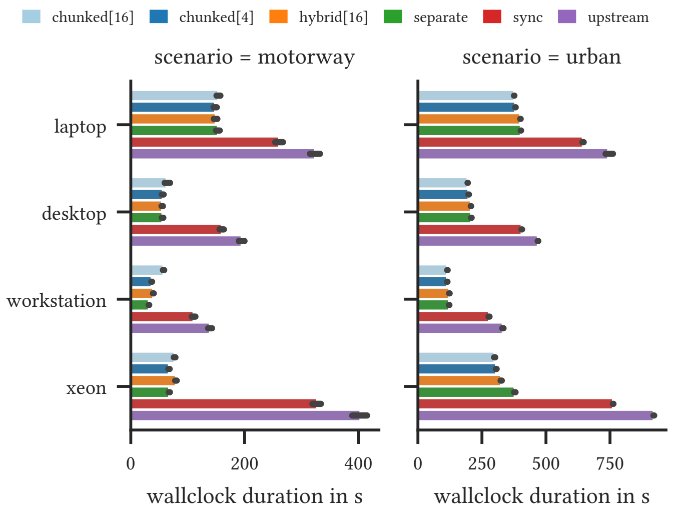


Figure 5: Simulation durations.

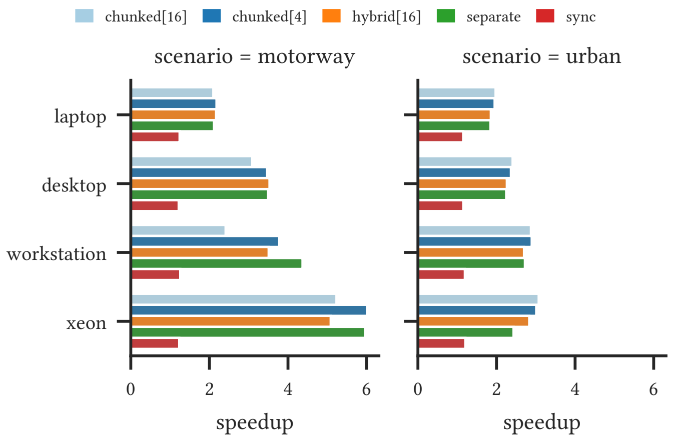


Figure 6: Simulation speedups.

6. Performance Evaluation

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 5 shows the mean durations for each scenario, platform, and parallel strategy, while Figure 6 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 5 also show the 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 laptop platform, the speedup for parallelized simulation lies at about 2, which matches its physical core count, with very little difference between the parallel strategies and scenarios. On the 4-core 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 the 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 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 workstation platform, the effects described for the 4-core 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 16-core xeon server platform shows the strongest impact of parallelization, due to its many, but individually less powerful cores (the CPU was released in 2012). While the runtime of single-threaded *upstream* and *sync* code is even longer than on the 2-core laptop, all parallel strategies show speedups of 5–6 in the motorway scenario. In the urban scenario, the effect is less pronounced, and the *chunked* strategies perform better than *hybrid* or *separate*, but still shows the larges speedup across all platforms.

The amount of CPU seconds spent in user mode (brighter, bottom bars) and kernel/system mode (darker, top bars) shown in Figure 7 reinforce these findings. The *separate* strategy is very efficient on powerful platforms and complex

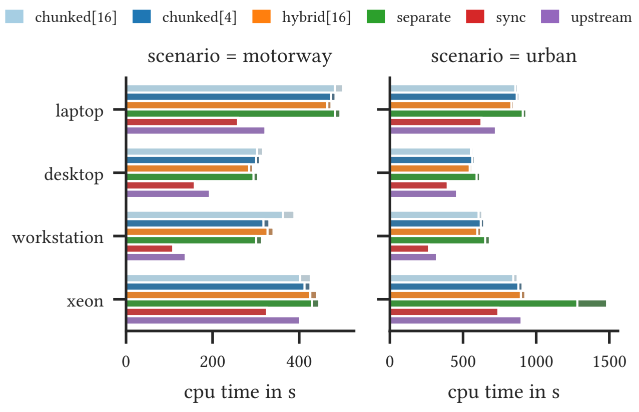


Figure 7: Total CPU (user) time spent during simulation.

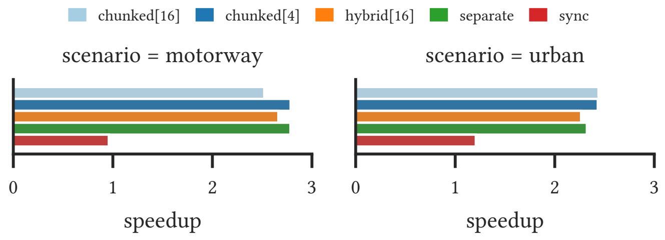


Figure 8: Simulation speedups for debug-builds on the 4-core desktop platform.

models but incurs more computation and synchronization overhead for less complex models. This ise especially pronounced on the 16-core xeon server platform for the urban scenario, where high CPU times in kernel mode are observed. The *hybrid* strategy remains the most efficient one in all cases but the motorway scenario on the 8-core workstation platform. But Figure 7 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 workstation platform is higher than on the 4-core 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. The 16-core xeon server platform is especially interesting in that regard: Total CPU time in single-threaded execution is roughly on par with the parallel versions, despite very different wall clock times (see Section 6.1), indicating very efficient parallel processing.

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.

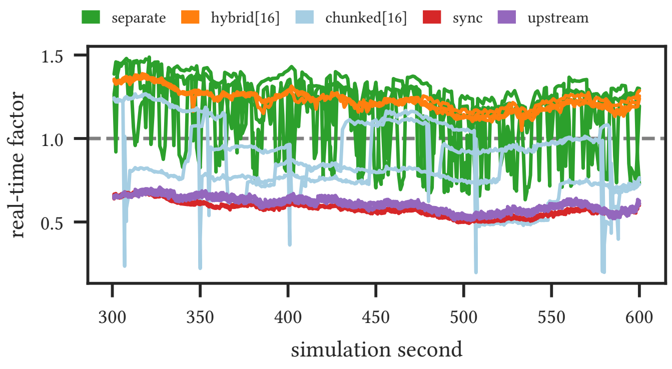


Figure 9: Real-time performance over simulation time on the 4-core desktop platform for the real-time motorway scenario.

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 desktop platform are shown in Figure 8. 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 6). 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. We configured it to produce values every 100 ms, a rate similar to synchronization intervals with real-time systems, e.g., the ego vehicle interface (EVI) [6]. 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`). This reduces the computational complexity far enough that the 4-core desktop platform can run the simulation in real-time.

Figure 9 shows the real-time factor for various parallel strategies across the simulation time on the 4-core desktop platform. Lines of the same color represent different runs of the exact same configuration. Both the *separate* and *hybrid[16]* strategies are able to run the simulation time in less than that in wallclock time. But the *separate* strategy has a much more volatile profile and fell below the real-time factor of 1.0 multiple times. The *chunked[16]* strategy is even more volatile with spikes way below the single-threaded strategies, thus proving inadequate for this use case. So, the *hybrid[16]* strategy appears to be more suited for real-time applications.

6.4. Discussion

The best strategy depends on the scenario, hardware platform, settings, and requirements—e.g., real-time stability. The *hybrid* strategy is the most versatile: Reasonably fast across all tested scenarios, efficient, and the most stable for real-time applications. However, other strategies may yield better performance in some regards and depending on the platform. E.g., *separate* on powerful CPUs like on 8-core workstation or many cores like on 16-core xeon server. We thus recommend starting out with *hybrid* and perform a few benchmarks for a given scenario to see if there are further speedups to be gained. As a rule of thumb, stronger platforms and more complex tasks (e.g., attenuation models) benefit from more fine-grained strategies, such as *separate*. This is especially the case for less powerful individual cores, as seen on the 16-core xeon server platform. 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 number 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.

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. Parallelization Evaluation

Now that we have seen the speedups of our implementation of ABP could achieve (see Section 6), one question remains: How much of the potential speedup was actually realized? To investigate this, we look at the improvements that were achieved from three perspectives: First, we compare them to the theoretical limits of an ideal implementation. Second, we profile the main thread of the parallelized code to investigate which portions of the code achieved how much speedup. Third, we perform and analyze advanced measurements on where the potential speedups were lost in practice due to parallelization overheads. Together, this provides a way to judge the effectiveness of the parallelized implementation and provides starting points for further optimizations.

7.1. Theoretic Limits

To assess how much speedup of the overall simulation can be gained through parallelization of the discovered potentials (see. Section 4.1), theoretic models can give some first insights. This yields upper bounds for the speedup, which correspond to perfect parallelization under ideal circumstances. Such bounds can be useful both before applying parallelization (to gauge if the effort may be worth the work) and afterwards (to gauge how much potential was actually achieved).

The speedup through parallelization can be estimated using Amdahl's law [35], as shown in Equation (1). It gives the theoretical speedup $S_{latency}$ of a whole task consisting of a sequential fraction $1-p$ and a parallel fraction p, with a given speedup s for the parallel fraction.

$$S_{latency}(s) = \frac{1}{(1-p) + \frac{p}{s}} \tag{1}$$

We can use this formula to gauge the effectiveness of parallelization through ABP. However, there are a few deviations to consider. First, all forms of overhead are ignored for this ideal speedup (or need to be incorporated into p). Second, Amdahl's Law assumes that each portion of the program is either sequential or parallel. But through ABP, background tasks can run while the main thread—a sequential portion of the program—continues to run.

We can find an upper limit of the speedup of this effect by assuming infinite parallel resources, i.e., $s = \infty$. Thus, in the ideal case for ABP, all background tasks are finished before the main thread requires their results., i.e., $\frac{p}{s} = 0$. By applying this to the portion of code to parallelize (see Section 4.1 and Figure 10), we receive the ideal speedup shown in the bottom row of Table 2. Note that these numbers may vary for different platforms, but still give a useful estimate.

Furthermore, we can still use Amdahl's Law to compare implementation to an ideal implementation of plain parallelization of the attenuation models, but without background processing. I.e., by using approaches like *OpenMP parallel loops* or thread pools to compute all attenuation models in parallel on message transmission, but *not* have the main thread continue at the same time. This would fully match the assumptions of Amdahl's Law. By using the same values for p as above and setting s to the number of threads available on our platforms, we get the values shown in Table 2. Note that this assumes full-fledged hardware threads, which requires more complex hardware than is implemented on typical processors with simultaneous multithreading (SMT).

Threads ($= s$)	Motorway ($p = 0.896$)	Urban ($p = 0.757$)
2	1.812	1.610
4	3.050	2.316
8	4.633	2.966
16	6.256	3.450
∞	9.631	4.124

Table 2: Theoretical ideal speedups for non-background parallelization of the attenuation models of Veins. Values of p collected on the 8-core workstation platform.

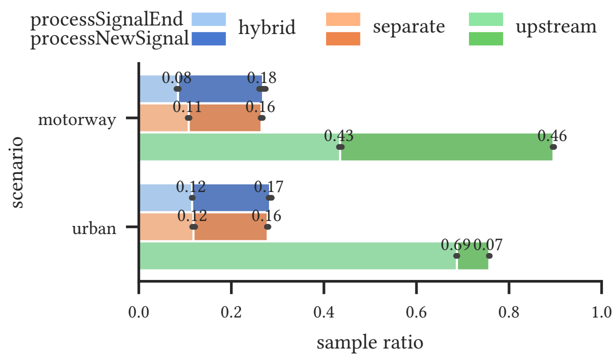


Figure 10: Proportion of main-thread samples in parallelized signal-processing functions (collected on the 8-core workstation platform).

7.2. Remaining Sequential Code

Now that we know how much speedup would have been possible under ideal conditions, we take a closer look at how much of it we actually achieved. To do so, we apply the profiling techniques from Section 4.1 again, but this time on the parallelized implementation. More precisely, on the main thread of the parallel code. This tells us how much of the functions we selected for parallelization were actually pushed the background threads and thus removed from the main thread runtime. For simplicity, we will perform this analysis on the 8-core workstation platform only, but the procedure is the same on all platforms.

Looking at the performance from the outside, Table 2 shows what the ideal speedup for the 8-core workstation platform would be. Depending on whether we consider only full cores or hardware threads, the values range between 4.633–6.256 for the motorway scenario and 2.966–3.450 for the urban scenario. But in the benchmarks (cf. Section 6.1), we only saw 4.370 with the *separate* strategy on the motorway and 2.896 using the *chunked[4]* strategy on the urban scenario. So, the practically achieved speedup at least gets close to the ideal case for the full core count of the platform. But it did not achieve anything near the potential of the $\frac{p}{s} = 0$ case that might have been possible for ABP.

This is due to two portions of code that are still executed sequentially. On one hand, there is the portion of code that is within the call tree designated for optimization during the potential search (cf. Section 4.1) but was not suited for parallelization. E.g., for Veins, we picked the `processNewSignal` and `processSignalEnd` functions as starting points for the parallelization effort. But not all code within those functions was parallelizable, e.g., due to interactions with the simulation kernel. On the other hand, there is new sequential code that was introduced as part of the parallelization effort. E.g., for Veins, we needed to collect all data needed for the background execution of the attenuation models, e.g., vehicle positions. This data collection needs to interact with the simulation kernel and thus cannot be offloaded to the background itself.

By running the profiling from Section 4.1 again, we obtain a breakdown of where the main thread still spends its time in the parallelized implementation. Figure 10 shows the proportions of the samples within the functions designated for parallelization. Around 26–28 % of the main thread is still spent within these functions — with little difference between the scenarios. While in the original upstream veins code, it made up a much larger portion of the runtime. And, of course, the total runtime was much longer in the upstream code than in the parallelized versions.

7.3. Parallel Code Efficiency

So in terms of Amdahl's Law, a significant factor for the sub-optimal speedup is due to a lower value of p, the portion of code that was actually parallelized. However, in practical applications, the value of s in Amdahl's Law will also not be equal to the number of threads or cores. Due to various overheads and imperfect distribution of work, this efficiency value will actually be lower. But the exact reasons are hard to investigate with the profiling techniques employed so far. So next we will turn to more advanced profiling techniques to find out where the wall clock time is spent in multi-threaded applications.

To investigate the multi-core behavior and overheads, we are using the *perf* tool introduced in Section 4.1 in an advanced technique: *Off-CPU profiling* [26]. In Off-CPU profiling, the profiling program (i.e., *perf*) instruments the kernel to discover context switches of the program under investigation. Whenever a context switch occurs, the current call stack is collected (as in normal sampling operation) and saved with the current time. And when the context is restored, the time is saved again. This allows three derivations:

1. deciding *why* a context switch happened by examining the switch event itself.
2. locating *where* the context happened in the code by following the call stack.
3. deriving *how long* the context switch took by comparing the time of the switch-out event with the following switch-in event of a particular thread.

Applying these three derivations on a certain thread provides information on how long (in wall-clock time) the thread was not actively running on the CPU. Thus, the name Off-CPU profiling. This circumvents the problem that in normal sampling operation, *perf* cannot sample a thread that is Off-CPU as it is not running.

By applying this to the main thread of the simulation, we can investigate where the ABP implementation loses efficiency. Because whenever the main thread spends time Off-CPU the simulation progress halts and while program duration increases. Ideally, the main thread should never experience any Off-CPU time. However, there are a number of reasons why this may be the case in practice. The most important ones are:

- blocked time waiting for network and other I/O. This may happen as the main thread waits for coupled simulators (e.g., SUMO) or writes results to disk. Such times cannot be optimized away for the main thread through parallelism. But with ABP, they can potentially be used by background threads to compute offloaded tasks.
- blocked time waiting for locks. This happens as overhead resulting from locking mechanisms to synchronize with worker threads. If a result is not available by the time it is required by the main thread, this leads to a stall. But even if all results are available on time, there is still the overhead of the synchronization primitive itself. The difference can be made visible by the duration of the context switch.
- waiting time while all CPUs are busy. This may happen if all cores are busy, e.g., with worker threads (or even other programs). In such cases the operating system's scheduler may suspend the main thread for fair allocation of resources to all threads. This typically indicates that the simulation runs as fast as it can on the given platform but could benefit from more CPU resources.

To perform Off-CPU profiling with perf, at least two special settings have to be activated: the `sched:sched_switch` events type has to be added and the `--switch-events` setting turned on. Together, these collect the data needed to reconstruct why, where, and for how long context switches occurred. However, in the initial data file produced by `perf record` the are still scattered and have to be combined by a later run of `perf inject --sched-stat`. In addition, we also enabled the `cpu-clock` event for profiling, which gives us the time spent in each call stack (in contrast to sample counts). This allows cross-checking if the On-CPU time and Off-CPU time of the main thread add up to the program runtime. Note that this Off-CPU profiling configuration is more resource intensive. It may require long post-processing and writes a lot of data (up to 50 GiB for our scenarios). We thus performed it on the more powerful 8-core workstation platform.

The resulting distributions of On-CPU and Off-CPU times of the main thread are shown in Figure 11. The sum of On-CPU and Off-CPU time yields the total runtime,

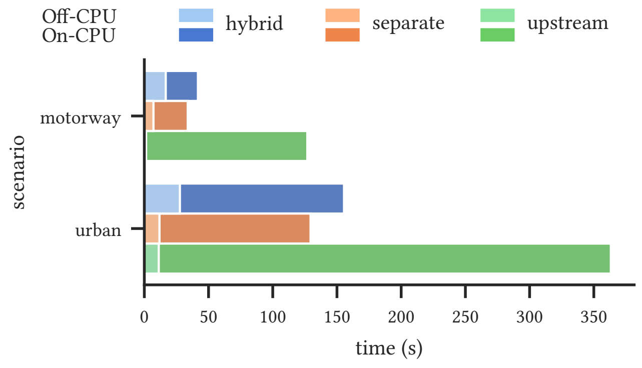


Figure 11: On-CPU vs Off-CPU times.

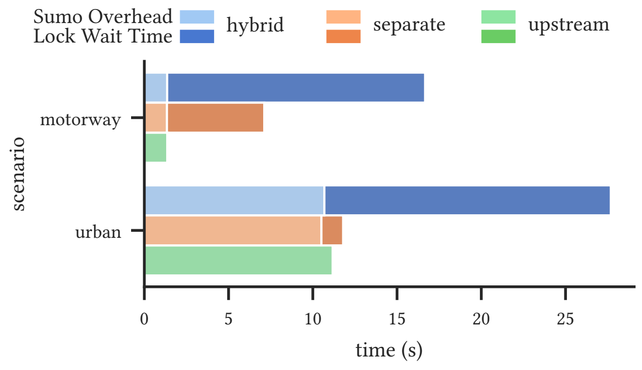


Figure 12: Details of Off-CPU times.

which matches with previous measurements without profiling. For the unmodified *upstream* code, there is very little Off-CPU time, as was expected. But for the parallelized *separate* and *hybrid* strategy, a large portion of their difference in total runtime is due to the off-CPU portion. This matches our previous observation (cf. Figure 10).

To gain a deeper insight on what lead to the Off-CPU times, we break them further down, as shown in Figure 12. As discussed above, we see overhead due to interaction with SUMO (blocked time waiting for network and other I/O), and locking (blocked time waiting for locks). Waiting time while all CPUs were busy was negligible on the 8-core workstation platform and is thus omitted in Figure 12. We can now see that the Off-CPU time of the *upstream* code is completely due to SUMO. In particular, there is a precise 1 s sleep time when Veins waits for SUMO to start up. It is present in both scenarios and makes up most of the Off-CPU time for the *upstream* code in the *motorway* scenario. The remaining 0.35 s of SUMO overhead stable across all versions of the *motorway* scenario, are due to the time when SUMO computes and sends the mobility updates for the vehicles. These mobility updates are much more pronounced in the *urban* scenario, where 9.68–10.18 s are needed to simulate the much larger number of vehicles and more complex traffic scenario. The actual runtime differences between the two parallel versions are due to the time their main threads spend waiting on locks, i.e., waiting for results from background worker threads. With only 1.18 s vs. 16.744 s of Off-CPU waiting time the *separate* strategy outperforms the *hybrid* strategy in the *urban* scenario. In the *motorway* scenario, with 5.56 s vs. 15.12 s respectively, the situation is similar, yet less pronounced.

So, Off-CPU profiling has provided useful insights for the understanding and further optimization of parallelized implementations and the application of ABP. The Off-CPU times show how the more aggressive *separate* strategy can utilize the powerful CPU of the 8-core workstation platform with a high number of individually strong cores. With this knowledge, a more balanced strategy like *hybrid* could be tuned for this platform, e.g., for real-time applications. Because as shown in Section 6.3, *hybrid* has benefits in terms of stable response times. Though these outcomes may look very different for other platforms or scenarios.

We have also seen how non-parallelized portions of the code dominate the remaining runtime. And external factors, e.g., waiting for SUMO to simulate traffic, are mostly unaffected by parallelization of computation tasks. On one hand, this puts the achieved runtime speedups, as seen in Section 6.1, into perspective. The possible improvement of total runtime is naturally limited, and the effectiveness of parallelization may seem smaller than it actually is. On the other hand, this provides directions for the next round of optimizations and parallelization.

8. Conclusion

We studied the importance of fast simulation tools for wireless communication and how Asynchronous Background Processing (ABP) can help accelerate these in situations where traditional parallelization approaches struggle. Without sacrificing accuracy and without changing the simulation model itself, our new concept can be applied to various simulation tools and target different portions of the code, running more efficiently the more work can be offloaded to background tasks. In an extensive case study, we have shown the application of in a realistic simulation example. We covered the complete process from parallelization potential analysis to implementation challenges, result verification, black-box performance evaluation, and detailed performance analysis of the resulting multi-threaded adaptation. Across multiple hardware platforms and in two different scenarios, ABP reliably achieved speedups of up to 3.5 on a typical desktop platform. Particularly real-time applications, which cannot utilize process level parallelism and need stable speeds across longer simulations, benefit greatly from the continuous speedup.

In future work, the concept could be applied to other parts of the simulation or to different simulation cores such as ns-3. We also aim to integrate our implementation into the next release of Veins so that the whole research community can benefit from faster simulations.

Acknowledgements

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] S. Joerer, C. Sommer, F. Dressler, Toward Reproducibility and Comparability of IVC Simulation Studies: A Literature Survey, COMMAG 50 (10) (2012) 82–88. doi:10.1109/MCOM.2012.6316780.

[2] A. M. Law, Simulation, Modeling and Analysis, 4th Edition, McGraw-Hill, 2007.

[3] C. Obermaier, R. Riebl, A. H. Al-Bayatti, S. Khan, C. Facchi, Measuring the Realtime Capability of Parallel-Discrete-Event-Simulations, Electronics, Communication Technologies for VANETs 10 (6) (Mar. 2021). doi:10.3390/electronics10060636.

[4] J. Xiao, P. Andelfinger, D. Eckhoff, W. Cai, A. Knoll, A Survey on Agent-based Simulation Using Hardware Accelerators, CSUR 51 (6) (2019) 131:1–131:35. doi:10.1145/3291048.

[5] R. M. Fujimoto, Research Challenges in Parallel and Distributed Simulation, TOMACS 26 (4) (May 2016). doi:10.1145/2866577.

[6] D. S. Buse, F. Dressler, Towards Real-Time Interactive V2X Simulation, in: IEEE VNC 2019, IEEE, Los Angeles, CA, 2019, pp. 114–121. doi:10.1109/VNC48660.2019.9062812.

[7] G. Kunz, M. Stoffers, O. Landsiedel, K. Wehrle, J. Gross, Parallel Expanded Event Simulation of Tightly Coupled Systems, TOMACS 26 (2) (Jan. 2016). doi:10.1145/2832909.

[8] D. R. Jefferson, Virtual time, TOPLAS 7 (3) (1985) 404–425. doi:10.1145/3916.3988.

[9] X. Zeng, R. Bagrodia, M. Gerla, GloMoSim: a Library for Parallel Simulation of Large-scale Wireless Networks, in: Workshop PADS 1998, IEEE, Banff, Canada, 1998, pp. 154–161. doi:10.1109/PADS.1998.685281.

[10] M. Gütlein, R. German, A. Djanatliev, Performance Gains in V2X Experiments Using Distributed Simulation in the Veins Framework, in: IEEE/ACM DS-RT 2019, IEEE, Cosenza, Italy, 2019. doi:10.1109/DS-RT47707.2019.8958671.

[11] D. S. Buse, G. Echterling, F. Dressler, Accelerating the Simulation of Wireless Communication Protocols using Asynchronous Parallelism, in: ACM MSWiM 2021, ACM, Virtual Conference, 2021, pp. 57–66. doi:10.1145/3479239.3485683.

[12] C. Sommer, R. German, F. Dressler, Bidirectionally Coupled Network and Road Traffic Simulation for Improved IVC Analysis, TMC 10 (1) (2011) 3–15. doi:10.1109/TMC.2010.133.

[13] Y. A. Sekercioglu, A. Varga, G. K. Egan, Parallel simulation made easy with OMNeT++, in: European ESS 2003, Delft, Netherlands, 2003.

[14] J. Pelkey, G. F. Riley, Distributed simulation with MPI in ns-3, in: ACM/ICST SIMUTools 2011, ACM, Barcelona, Spain, 2011, pp. 410–414.

[15] P. Peschlow, A. Voss, P. Martini, Good News for Parallel Wireless Network Simulations, in: ACM MSWiM 2009, ACM, Tenerife, Spain, 2009, pp. 134–142. doi:10.1145/1641804.1641828.

[16] I. Al-Shiab, A. Sabbah, A. Jarwan, O. Issa, M. Ibnkahla, Simulating large-scale networks for public safety: Parallel and distributed solutions in NS-3, in: IEEE PIMRC 2017, IEEE, Montreal, Canada, 2017. doi:10.1109/PIMRC.2017.8292761.

[17] A. Sabbah, A. Jarwan, I. Al-Shiab, M. Ibnkahla, M. Wang, Emulation of Large-Scale LTE Networks in NS-3 and CORE: A Distributed Approach, in: IEEE Milcom 2018, IEEE, Los Angeles, CA, 2018, pp. 493–498. doi:10.1109/MILCOM.2018.8599762.

[18] G. Kunz, O. Landsiedel, S. Götz, K. Wehrle, J. Gross, F. Naghibi, Expanding the Event Horizon in Parallelized Network Simulations, in: IEEE MASCOTS 2010, IEEE, Miami Beach, FL, 2010. doi:10.1109/MASCOTS.2010.26.

[19] I. Mavromatis, A. Tassi, R. J. Piechocki, A. Nix, Poster: Parallel Implementation of the OMNeT++ INET Framework for V2X Communications, in: IEEE VNC 2018, Poster Session, IEEE, Taipei, Taiwan, 2018. doi:10.1109/VNC.2018.8628429.

[20] P. D. Barnes, M. D. Bielejeski, D. R. Jefferson, S. G. Smith, D. G. Wright, L. Giupponi, K. Koutlia, C. Harper, S3: the Spectrum Sharing Simulator, in: 2019 WNGW 2019, ACM, Florence, Italy, 2019, pp. 34–37. doi:10.1145/3337941.3337945.

[21] C. Obermaier, R. Riebl, C. Facchi, A. H. Al-Bayatti, S. Khan, COSIDIA: An Approach for Real-Time Parallel Discrete Event Simulations Tailored for Wireless Networks, in: ACM SIGSIM PADS 2021, ACM, Virtual, Online, 2021, pp. 165–171. doi:10.1145/3437959.3459250.

[22] D. Eckhoff, A. Brummer, C. Sommer, On the Impact of Antenna Patterns on VANET Simulation, in: IEEE VNC 2016, IEEE, Columbus, OH, 2016, pp. 17–20. doi:10.1109/VNC.2016.7835925.

[23] F. Bronner, C. Sommer, Efficient Multi-Channel Simulation of Wireless Communications, in: IEEE VNC 2018, IEEE, Taipei, Taiwan, 2018. doi:10.1109/VNC.2018.8628350.

[24] R. M. Fujimoto, Time Management in The High Level Architecture, SIMULATION: Transactions of The Society for Modeling and Simulation International (SIMULATION) 71 (6) (1998) 388–400. doi:10.1177/003754979807100604.

[25] R. H. Halstead, MULTILISP: a language for concurrent symbolic computation, TOPLAS 7 (4) (1985) 501–538. doi:10.1145/4472.4478.

[26] B. Gregg, The Flame Graph, Communications of the ACM 59 (6) (2016) 48–57. doi:10.1145/2942427.

[27] T.-W. Huang, C.-X. Lin, G. Guo, M. Wong, Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++, in: IEEE IPDPS 2019, IEEE, Rio de Janeiro, Brazil, 2019, pp. 974–983. doi:10.1109/IPDPS.2019.00105.

[28] D. S. Buse, Experiment Setup for "Accelerating the Simulation of Wireless Communication Protocols using Asynchronous Parallelism", Simulation Experiment Setup version 1.0, Zenodo (9 2021). doi:10.5281/zenodo.5503502.

[29] M. Vincent, F. Berens, Survey on ITS-G5 CAM statistics, TR 2052, V1.0.1, C2C-CC (Dec. 2018).
URL `https://www.car-2-car.org/fileadmin/documents/General_Documents/C2CCC_TR_2052_Survey_on_CAM_statistics.pdf`

[30] R. Molina-Masegosa, M. Sepulcre, J. Gozalvez, F. Berens, M. Vincent, Empirical Models for the Realistic Generation of Cooperative Awareness Messages in Vehicular Networks, TVT 69 (5) (2020) 5713–5717. doi:10.1109/TVT.2020.2979232.

[31] B. Bloessl, A. O'Driscoll, A Case for Good Defaults: Pitfalls in VANET Physical Layer Simulations, in: IFIP WD 2019, IEEE, Manchester, United Kingdom, 2019. doi:10.1109/WD.2019.8734227.

[32] I. Khan, J. Härri, Can IEEE 802.11p and Wi-Fi coexist in the 5.9GHz ITS band ?, in: IEEE WoWMoM 2017, IEEE, Macau SAR, China, 2017. doi:10.1109/WoWMoM.2017.7974358.

[33] C. Sommer, D. Eckhoff, R. German, F. Dressler, A Computationally Inexpensive Empirical Model of IEEE 802.11p Radio Shadowing in Urban Environments, in: IEEE/IFIP WONS 2011, IEEE, Bardonecchia, Italy, 2011, pp. 84–90. doi:10.1109/WONS.2011.5720204.

[34] C. Sommer, S. Joerer, M. Segata, O. K. Tonguz, R. Lo Cigno, F. Dressler, How Shadowing Hurts Vehicular Communications and How Dynamic Beaconing Can Help, TMC 14 (7) (2015) 1411–1421. doi:10.1109/TMC.2014.2362752.

[35] G. M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, in: AFIPS 1967 Spring Joint Computer Conference (AFIPS 1967), Vol. 30, ACM, Atlantic City, NJ, 1969, pp. 483–485. doi:10.1145/1465482.1465560.