# Multi-core parallelism for ns-3 simulator

Internship at INRIA Sophia-Antipolis, PLANETE team

Guillaume Seguin

15 June 2009 - 28 August 2009

## ABSTRACT

The ability to quickly evaluate network protocols or infrastructure shapes on real-life scale simulations is a key factor for network researchers work. Since multi-core systems are now widely available and installed, doing multi-threading parallelism is a simple but potentially efficient optimization for simulators performance, which may also be implemented so that the parallelism is completely transparent to the user. Consequently, we designed and implemented a seamless multithreaded simulator implementation for ns-3, a next-generation network simulator for Internet systems intended as a replacement of the popular ns-2. We explain here the details of this implementation and describe its development process, which includes stating on the design choices, making the simulator thread-safe and optimizing the implementation.

## Contents



# 1 PARALLELISM FOR NS-3

## 1.1 Our goals

This project aimed at implementing parallelism into ns-3 with two main goals in sight : speeding up simulation times while being as little intrusive for the end user as possible.

We chose to do multithreaded parallelism instead of distributed parallelism, *i.e.* doing parallelism using the resources of a single computer rather than using a bunch of computer. This choice is governed by two facts : first, having to set up several computers for the simulation is obviously heavily intrusive for the user ; second, distributed algorithms would require a static partitioning of the simulation (*i.e.* of the network in the case of ns-3), which we neither want to ask the user for nor want to compute.

Furthermore, we do not have any goal of scalability (*i.e.* we do not specifically want to be able to keep simulation times at the same order of magnitude while increasing the size of the simulated network by orders of magnitude, which we could do with distributed parallelism where it would only be a matter of adding more machines to the computation cluster), we only want to improve simulation times as much as we can using a single machine.

The will of user-friendliness lead to a number of other design choices which are detailed in 2 : 'Doing seamless parallelism'. The performances optimizations are described in 3 : 'Doing efficient parallelism'.

## 1.2 About ns-3

ns-3 is a discrete-time, event-driven network simulator, which means that it simulates timestamped events, so that the internal clock of the simulator (simulation time) advances in a discrete manner. It aims network researchers working on Internet systems, and attempts to ease the reuse of existing code for networking protocols, networking stacks or applications.

ns-3 is a free, open-source software, licensed under GNU GPLv2, written in C++ and offering Python bindings. Development of ns-3 was started in 2006, and releases are made about every 3 months or so. ns-3 is meant to become a replacement for ns-2, a popular network simulator written in C++ and OTcl, which suffers from some design issues.

### 1.2.1 *Simulator architecture*

ns-3 is split between a core simulator part and a models part.

The simulator part is just a discrete-time event simulator, to which the rest of the code gives events to process at a given time. The simulator implementations do not know anything about what they actually process, not even that they simulate networks or even just graphs.

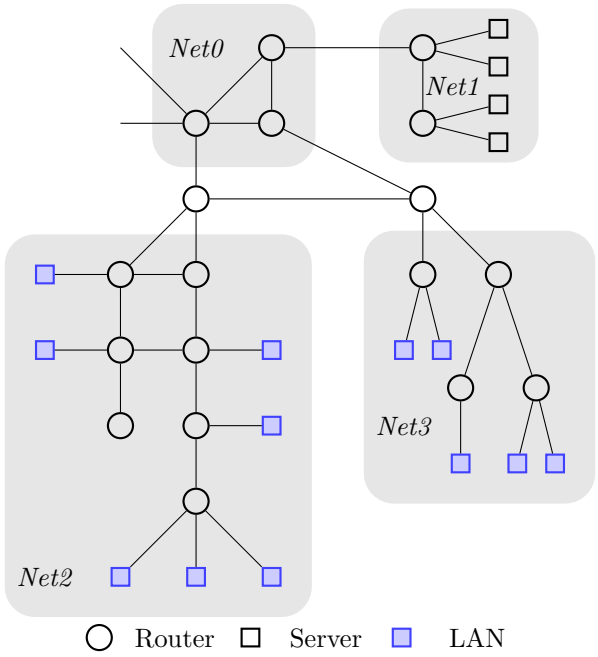
This knowledge is held in the models part of ns-3. These models specify the type of network topology (*e.g.* a networks with nodes and channels between nodes), the devices behaviour (*e.g.* CSMA or 802.11 interfaces), the mobility behaviour (for moving nodes, such as laptops connected through wireless) and the applications that run on the nodes.

The network itself is specified in a Python or C++ script, as well as the applications running on it.

One of the main issues with implementing a multithreaded simulator for ns-3 is to make it as little intrusive as possible, both for the models part as well as for the script.

## 1.3 Test case

During the development of this multithreaded simulator implementation, we used a classic parallel networking simulation test case, which is based on the DARPA NMS Campus Network model presented in [Nicol, 2003]. The simulated topology is a ring of interconnected campus networks. Each campus network is split into 4 subnetworks, labeled *Net0* to *Net3*.




*Net0* is the backbone of the network, with a border router linked to other campus networks and routers linked to the other subnetworks ; *Net1* is the campus servers subnetwork ; *Net2* and *Net3* are client subnetworks, with each a given number of LANs of 42 clients.

# 2 DOING SEAMLESS PARALLELISM

The goal of making the multithreading implementation as transparent as possible to the user lead to a number of design choices : how should the network be partitioned, what algorithms should be used, how should thread-safety be handled ?

## 2.1 Partitioning the network

A successful parallel application heavily relies on the appropriate distribution of the workload over the execution units (*i.e.* the threads in our case). Yet, we definitely do not want to ask the user to define a static partitioning of his network, since we want the multithreading part to be as little intrusive as possible, neither do we want to precompute a static partitioning of the network based on the single topology knowledge, which does not hold enough information to produce an efficient partitioning (the correct partitioning at some point can become much less efficient at some other point in the future as data flows change). Moreover, in either case the partitioning would depend on the number of threads, which adds another complexity layer.

Consequently, we use an extreme version of the partitioning idea where each node is a partition. This way, we will be able to relocate partitions over execution units as required during the simulation to dynamically balance the workload.

*Identifying partitions* Inside the multithreaded simulator implementation, each partition is represented, by a clock and a queue of events called the *scheduler*. One of the problems that had to solve at the API level was that events were previously scheduled on a single queue of events, so that we had to find a way to identify the target partition (and so the target events queue) when scheduling events. Since the simulator part of ns-3 has no knowledge of what it simulates, we added the notion of *context*. This notion is not bound to the fact that we are simulating networks, and can also be useful for logging and debugging purposes. It is used this way : when scheduled, each event is associated to a context, which was either explicitly specified (*i.e.* by the user or the model from which the event is scheduled) or automatically inferred (*i.e.* an event scheduled while processing another event will be given the same context as the event being processed if the context is not specified otherwise). In the multithreaded simulator, this context is interpreted as the partition identifier, which is used to put the event into the appropriate event queue. Outside of the simulator part, this is simply the node identifier.

## 2.2 Synchronization algorithm

The next step after partitioning the network is to choose a synchronization algorithm, which is the guarantee that the simulation is run in a deterministic and consistent manner. What needs to be ensured is a simple but strict ordering on the dates of the events processed on each node of the network. Thus, the synchronization algorithm task is to define, at each iteration, the maximum date up to which each partition clock (separately) can advance during the iteration, so that the ordering constraint is respected.

Two main kinds of synchronization algorithms exist : *conservative* and *optimistic* algorithms.

Conservative algorithms take the approach that the best thing to do to handle desynchronization is just to avoid it, so that such algorithms strictly enforce the ordering constraint. The downside of this approach is that it can lead to situations where only a few events are processed on each execution unit during each iteration of the algorithm, so that the overhead of the algorithm cannot even be balanced by the speed improvement brought by parallelization.

On the opposite, optimistic algorithms assume that the desynchronization points are rare and that the gain of letting the executions units process events at full speed is well worth the cost of interrupting the simulation sometimes and fixing consistency errors.

The main problem with the optimistic approach is that it is hard to implement. Three paths can be used to do the recovery :

- Ask the users to provide an Undo function matching each event they define, function which will be used to revert all the events that lead to the consistency faults. This approach is definitely not possible since it really intrusive and would require deep rework of the currently existing models codebase.
- Write a specific compiler which would produce the Undo functions : this is more than non-trivial and probably really hard to implement.
- Frequently take snapshots of the status of the whole network (which mostly means taking snapshot of the whole simulator memory) and restore them if anything goes wrong. This is most likely a very expensive approach time-wise and memory wise (because of the need to backup the whole memory) and finding the right frequency at which snapshots should be taken is a non trivial problem (the cost of taking the snapshots has to be balanced with the cost of the work to redo when the simulation goes wrong).

Consequently, we chose to use conservative synchronization algorithms. While working on this project, we have implemented two algorithms : the Chandy-Misra-Bryant algorithm and a synchronization-barrier-based variant.

*Algorithm basic shape* The basic shape of the simulator execution main loop is the following :

```
while simulation not finished:
    for each partition P:
      P.max_date = synchronization_algorithm (P)
    for each partition P:
      for each event message M:
        append M.event to P events queue
      while P.next_event_date < P.max_date:
        process (P.next_event)
```

We note here the use for each partition of a queue of incoming event messages, which are messages sent from neighboring partitions when they schedule an event on the current partition (most likely a packet receive event scheduled through a channel).

The two iterations on the partitions set can be easily parallelized, for instance by having a global thread-safe shared list of partitions from which each thread picks partition until the list is empty, at which point one of the threads refills it.

### 2.2.1 Chandy-Misra-Bryant algorithm

The Chandy-Misra-Bryant synchronization algorithm ([Chandy and Misra, 1979], [Bryant, 1977]) is a classic, well-known, algorithm which has been studied and used for all kinds of parallel applications.

This algorithm is based on the notion of *lookahead*, which is the minimum transmission delay from one partition to another. At a given iteration, the maximum date up to which the events of a partition can be processed is defined as the maximum date before events created from neighboring partitions may have to be processed. Since the events created from neighboring partitions have to go through a transmission channel, the computation of this maximum date just means computing the minimum of, for each neighboring partition, the neighboring partition clock + the lookahead from the neighboring partition to the current partition.

The correctness of the algorithm is quite obvious : since during a given iteration the events which can be processed are those which date is strictly lower than the minimum date of any event that could be scheduled from outside the partition, the simulation is guaranteed to be consistent.

The communication between the partitions is made using *null-messsages* : when a partition has finished processing its events, it sends such a message to the neighboring partition, telling them about the new value of the partition clock. Before computing the new maximum date of a given partition, the synchronization algorithm first processes those messages, updating a

local memory of the neighboring clocks, which is then used for the computation.

Thus, the synchronization algorithm is :

```
synchronization_algorithm (P):
  for each stored null-message M:
    P.neighbor_clock[M.sender] = M.clock
  max_date = ∞
  for each partition P2 in P neighborhood:
    max_date = min (max_date,
                    P.neighbor_clock[P2]
                      + P.lookahead[P2])
  return max_date
```

A little trick which is required for this algorithm to work is that each partition clock has to be set to the maximum date at the end of the iteration to be able to solve cases where would be no event in the time frame $[clock, max\_date[$, which may eventually lead the simulation to a state where all the partitions are waiting for a single partition which has no event scheduled until a date relatively far in the future and which clock is frozen because it cannot advance using the events.

This algorithm, yet being quite simple, suffers from two design issues.

*0-lookahead cycles* The first issue arises when an oriented cycle of partitions with 0 lookahead links between them. Since each partition can advance up to the clock of the previous partition in the cycle + 0, and since it is a cycle, all the clocks are bound to remain 0 forever. Obviously, a cycle of partitions with 0 lookahead is not physically feasible, a lookahead of 0 could be used as an approximation in some models where computing the real value or a non-0 approximate would be too expensive.

Such situations could probably be broken by some tricky heuristics, that is, by advancing the clocks manually according to the other neighboring clocks and if a tie exists between events (if two events were scheduled for the same date on two partitions of the cycle) decide which event to process first according to an extra parameter, such as the lowest partition identifier, but this would make the algorithm way more complicated and would heavily increase the synchronization overhead (detecting that the problem exists would probably be even more expensive than fixing it).

*Cost of the null-messages process* The second issue is that the time advance of the algorithm is bound to the use of the null-messages. That is, if the maximum lookahead is $L$, the simulation will only advance of $L$ time-wise at each iteration. Let's imagine that between the dates 0 and $T$ there is no event for any of the $n$ nodes of the network. To advance to the first event it will take $T/L$ iterations and $n * T/L$ null-messages. For 1000 nodes, $T = 1$s and $L = 10$ms, this means 100 iterations and 100000 null-messages for no actual simulation work. Even in less specific situations, the fact that advancing the simulation time is expensive when there is no event is a real performance issue which cannot be balanced by the use of multiple cores.

### 2.2.2 Barrier-based algorithm

Both problems of the Chandy-Misra-Bryant algorithm are related to the fact that only the clocks of the partitions are taken into account, and not the actual events waiting to be processed. Thus, a smarter algorithm is to compute a global maximum date based on the next event of each partition and the minimum lookahead from this partition to the neighboring partitions. Note that the meaning of the lookahead is the opposite of its meaning in the Chandy-Misra-Bryant algorithm, where the lookahead was the minimum transmission delay from the neighboring partition to the current one.

This algorithm, described in [Fujimoto, 2000], is based on synchronization barriers, which are meeting points which all threads must reach before any of them can continue. The idea is that at each iteration all the threads reach such a barrier, then one or all of them compute the global maximum date as specified above, then they all reach another barrier, then they do the events processing.

Here is how an iteration of the algorithm looks, codewise :

```
wait at barrier
if thread_id == 0:
  max_date = ∞
  for each partition P:
    max_date = min (max_date,
                    P.next_event_message.date
                      + P.lookahead,
                    P.next_event.date
                      + P.lookahead)
wait at barrier
for each partition P:
  for each event message M:
    append M.event to P events queue
  while P.next_event_date < max_date:
    process (P.next_event)
```

This algorithm definitely solves both problems of the Chandy-Misra-Bryant approach : no CPU time is spent on advancing the clock without doing any actual work, and the 0-lookahead cycles problem is solved since the computation of the maximum date ensures that at least the event with the minimum date of the whole simulation will be processed.

The correctness of the algorithm is once again quite simple : for each partition, the maximum date is defined as at most the minimum date at which an event may be scheduled on a neighboring node, so it ensures that each partition will not lead to a consistency problem. Thus, the algorithm is correct.

Performance wise, as stated above the algorithm does not create extra work for no actual simulation work. Yet, it might be less efficient than the previous algorithm since the computed maximum date is likely to give much smaller time frames for each partition at each iteration (the previous algorithm gave maximum dates which were computed for each partition while this one gives a global date based on similar local considerations). Furthermore, the use of synchronization barriers is a crucial point since it requires that the workload is well balanced over the threads to reduce the time wasted at waiting at the barriers.

This algorithm will be our primary focus in the remainder of this document and will be the subject of any comment or addition.

#### 2.2.3 *Dealing with global events*

The last issue that we could face is that for compatibility reasons the user may still schedule events outside of any context (from the `main ()` function of the script for instance) without specifying any context either. The problem is that we consequently have no idea of which node the event belongs to (it may even not belong to a node and may just be a time logging function or so) and as such we have no time/space constraint we could use to compute an appropriate lookahead.

Consequently, we chose to attribute a special context to these events (in the code it is `0xffffffff`, which $-1$ as an unsigned 32 bits integer) and place them in a list of global simulator events. This list, plus a corresponding global clock, is considered as a special partition, with a lookahead of 0 and processed from inside the safe place of the synchronization barrier : after computing the global maximum date for the normal partitions, noted `max_date`, we check the global events list, and if an event in the list is scheduled to occur before `max_date`, we update `max_date` to the event date, and process it (and any global event with the same date). The remainder of the algorithm is unchanged.

### 2.3 Transparent thread-safety

The structure of ns-3 is such that most data is only modified when the simulator is processing an event related to the node the data is linked to. That is, most data structures, such as network interfaces, ip addresses, networking stacks, are linked to a network node and are never used from events working on other network nodes.

Consequently, a simple way to avoid having to protect most of the data is just to ensure that two events related to a single node are not processed at the same time, property which is actually guaranteed by the synchronization algorithms for consistency reasons (modifying a node state must be done in a deterministic way, so parallel modifications of the state are excluded).

Thus, the only problem for these data structures is the fact that ns-3 uses reference counting for efficient and non intrusive memory handling, and that some objects linked to one node are sometimes accessed for reads from objects linked to another node. For instance, when a packet goes from one node to another one through a communication channel, the destination node identifier is read from the source node, which means that the reference counter of the destination node might has been accessed while processing an event not directly related to it, and so it might have been accessed from two threads at the same time, possibly losing some increments or decrements. Consequently, the reference counting system has to be made thread-safe where required, that is, for almost all reference counted objects in ns-3, or the reference counting system should never be used in code parts that may be ran from separate threads.

Disabling the reference counting for these objects when needed would require quite a bit of re-engineering of the models, so that they use raw pointers instead of smart pointers (*i.e.* magic objects that wrap real (raw) pointers, providing the same features as the wrapped pointers plus some additional features, such as automatic memory management), which removes the automatic calls to ref/unref operations. This re-engineering, which can lead to really tricky issues, is more expensive for the end user or the model developer than we can accept, so we will simply make the reference counting thread-safe and pay the performance price (though we will try to make it as low as possible).

The only data structures which do not belong to the previous category are the packet related buffers and caches, which are shared by all nodes and are used to optimize dynamic memory allocation and freeing for packet headers and metadata. The current solution for this issue is just to disable the said buffers and caches, so that this is a further work item (see 4.5 : 'Efficient packet-related caches').

## 3 DOING EFFICIENT PARALLELISM

Now that we have chosen the main lines of the implementation which ensure that the multithreaded simulator will not be too intrusive for the end user or the other contributors, we can focus on the other goal of this project : making simulations faster. We will describe the tools used to measure code performances and the design details for the synchronization barriers, reference counting and workload balancing.

### 3.1 Performance analysis tools

While working on the multithreaded implementation, we used several paths to analyse the behavior of the different parts of the code.

The most simple way we used to measure the performance of some parts of the code was by fetching the CPU clock before and after the considered code and doing the difference. This is an intrusive but simple and efficient way to benchmark parts of the code, and can even be used in some workload balancing heuristics.

We also used `Sysprof`[1], which is a system-wide profiler for Linux, which works by sampling callstacks of each process, giving statistical data on the time spent in each function. The great thing with sampling profilers is that the instrumentation is non-intrusive for the running code, as opposed to intrusive profilers such as `GNU gprof`[2], which instrumentation may heavily change the profiling results because of the overhead added to the instrumented code. `Sysprof` features a nice graphical user interface which makes it easy to read and sort the reports.

A large part of the performance analysis was done using `Oprofile`[3], which works just like `Sysprof` but is used through a command line line interface which allows much more customisation of the output. It can produce the same reports as `Sysprof`, but can also isolate some symbols and show where they are the most used or even annotate the source code showing where the time is spent.

The last main tool we used to benchmark the code was `Callgrind`[4], which is a profiling tool distributed with the `Valgrind`[5] suite. It provides information about callgraphs, CPU caches usage, memory references and executed instructions. Used in combination with `KCachegrind`[6], a graphical user interface for viewing `Callgrind` output, it is easy to quickly evaluate which parts of code lead to various performance bottlenecks.

## 3.2 Barrier implementations

One of the keys for successfully implementing this algorithm is the choice of the barrier implementation. Indeed, the barriers used in the algorithm need to be both reactive (extended sleeps, for instance, would lead to losing precious time) and cheap (doing expensive operations would also lead to losing some more precious time).

Thus, we have tested four different kinds of barriers to find the most efficient one.

*POSIX barriers* The easiest way to implement a synchronization barrier is to just use the implementation provided by the `pthread` library (which is itself part of the `glibc`). This implementation offers two operations :

- `init (n)`, which sets the number of threads which will be waited before any of them is released to `n`.
- `wait ()`, which blocks until `n` threads have reached the barrier.

*Semaphore-based barriers* The classic hand-made way to do a synchronization barrier is by using semaphores. Semaphores are specific protected counters aimed at parallel programming. Three operations can be done on a semaphore :

- `init (n)`, which sets the initial value of the semaphore to $n$.
- `post ()`, which atomically increments the semaphore value.
- `wait ()`, which blocks until the value of the semaphore can be atomically decremented while remaining positive (that is, until the semaphore becomes strictly greater than 0 and it is then successfully decremented by the operation).

The barrier itself is based on a master/slave hierarchy, with the master being predefined (for instance it could be the first computation thread). Two semaphores are used, `masterSemaphore` and `slaveSemaphore`, both initialized to 0. The idea is the master will first `wait` for masterSemaphore once per slave thread. Meanwhile, each slave will `post` masterSemaphore once, then `wait` at slaveSemaphore. Once master thread has been able to run `wait` for the number of slave threads, it is guaranteed that all the slave threads are waiting for the masterSemaphore, so that the barrier is reached by all the threads. The last thing to do for master is to release the slaves by doing `post` on slaveSemaphore once for each slave.

The following code summarizes the algorithm :

```
void
SemaphoreBarrier::MasterWait ()
{
  // Wait for the slaves
  for (i = 0; i < nThreads - 1; ++)
    masterSemaphore.wait ();
  // Synchronization point : barrier reached
  // Release the slaves
  for (i = 0; i < nThreads - 1; ++)
    slaveSemaphore.post ();
}

void
SemaphoreBarrier::SlaveWait ()
{
  // Tell the master we reached the barrier
  masterSemaphore.post ();
  // Wait for barrier termination
  slaveSemaphore.wait ();
}
```

A little trick is required to get a POSIX compliant behaviour, which is that the barrier can be safely reused immediately after being released. The problem with

[1] `http://www.daimi.au.dk/~sandmann/sysprof/`
[2] `http://sourceware.org/binutils/docs/gprof/`
[3] `http://oprofile.sourceforge.net/`
[4] `http://valgrind.org/info/tools.html#callgrind`
[5] `http://valgrind.org/`
[6] `http://kcachegrind.sourceforge.net`

this implementation is that for now, two successive calls of `SlaveWait ()` could succeed, as showed by the following instructions traces, with 3 threads :

```
Master          Slave 1         Slave 2
Call to Wait () Call to Wait () Call to Wait ()
master.wait ()
                master.post ()
                slave.wait ()
master.wait ()
                                master.post ()
slave.post ()
                slave.wait ()
                RELEASED
                Call to Wait ()
                master.post ()
                slave.wait ()
slave.post ()
RELEASED        RELEASED        slave.wait ()
```

Thus, the slave 1 has crossed two barriers, while master has only crossed one and slave 2 is still blocked at the first barrier.

To prevent this situation, the solution is to use two slave semaphores and switch the current one each time the synchronization point has been reached in the master thread and wait/post the correct slave semaphore in master and slave :

```
...
// Synchronization point : barrier reached
curSem = currentSlaveSemaphore
currentSlaveSemaphore = otherSlaveSemaphore
otherSlaveSemaphore = curSem
// Release the slaves
for (i = 0; i < nThreads - 1; ++)
  curSem.post ();
...

...
curSem = currentSlaveSemaphore
masterSemaphore.post ();
// Wait for barrier termination
curSem.wait ();
...
```

*Global spinlock based barrier* Another way to implement a synchronization barrier is by using a *spinlock*, which is an active loop continuously checking for a given condition, such as the availability of a given resource.

A simple way to implement such a barrier is by using an atomic counter, initially set to 0, a release bit initially set to 0 and a thread specific release bit, initially set to 1. The idea is that each thread getting into the barrier increments the counter, then loops until the global release bit and the thread specific one are equal. The last thread getting into the barrier detects that it is the last by checking the return of the counter incrementation, then resets the counter to 0 (to rearm the barrier) then flips the common release bit. The last thing to do is to flip the thread-local release bit of each thread right after leaving the loop. This way, the barrier will be ready for a new call.

If the barrier should be waiting for $n$ threads, the barrier code looks like :

```
void
GlobalSpinBarrier::Wait ()
{
  int old = AtomicExchangeAndAdd (&m_nWait, 1);
  if (old == n - 1)
  {
    m_waiters = 0;
    m_global_bit = !m_global_bit;
  }
  while (m_global_bit != m_local_bit)
  {}
  m_local_bit = !m_local_bit;
}
```

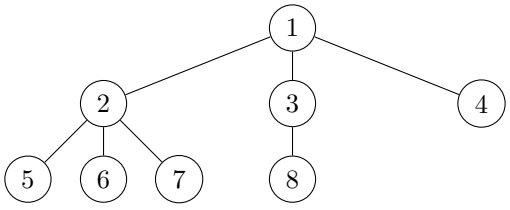
The good thing with this implementation is that it is active, while the two previous ones were passive (the `pthread_barrier_wait` and `sem_wait` calls were entering the operating system kernel, which usually leads into some sort of sleep state until the resource becomes available). Thus, the amount of time spent at waiting here should be almost reduced to the time lost because of the threads workload unbalance.

Yet, there is still a little performance issue, with the fact that the counter is shared by all the threads, so that each modification of the counter leads to the invalidation of the corresponding cache line for each core. Since with $n$ threads the counter will be incremented $n$ times, this leads to $n^2$ cache lines invalidations, which is quite a scalability problem since it is exponentially expensive as the number of threads increase.

*Tree shaped spinlock based barrier* .

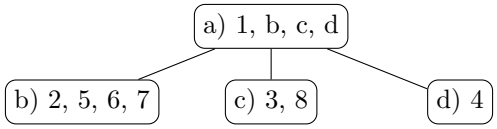
A simple solution to that problem is to limit the number of threads sharing a given resource to a predefined number. This is easily done by using a tree of spinlocks, where each thread is a node of the tree. The tree is built using the following mechanism : the first thread becomes the root of the tree, and the $m$ next threads become its children. Then, the $m$ next thread become the children of the second thread, and so on.

With 8 threads and $m = 3$, the produced tree is :




Each node and its immediate children share a single barrier similar to the global barrier of the previous paragraph. The actual dependencies tree for the previous

situation is, with each letter-indexed group corresponding to a single barrier :




The algorithm is basically the same as before, only difference being that when the last node of a group reaches the barrier, it first goes to wait at the barrier of the parent group before releasing the other nodes of the group.

For instance, with the previous example, we could have :

```
thread 8 enters wait
thread 4 enters wait => group d enters wait
thread 5 enters wait
thread 6 enters wait
thread 7 enters wait
thread 1 enters wait
thread 2 enters wait => group b enters wait
thread 3 enters wait => group c enters wait
group a released => groups b, c, d released
```

Code wise, there is not much difference with the global version. The `wait ()` function simply wraps a `wait_aux (node)` functions which waits at the given node barrier. The last node of a given group entering `wait_aux` will then call the `wait_aux` function with the parent node as argument, implementing the tree shape.

This implementation should thus fix the scalability of the spin-based barrier while not adding that much overhead to the original implementation.

The choice of the $m$ parameters (the number of threads in each group) is a strong factor for the success of this implementation. A low $m$ parameter could increase the algorithm overhead while a high one (relatively to the number of threads) could just make the algorithm pointless compared to the base algorithm. We benchmarked the performance of the barrier with various (threadscount, $m$) pairs by doing 10000000 successive calls to the `wait ()` function from each thread :

Table 1: Average barrier `wait ()` cost (in cycles)

| $m$ value | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 thread | 65 | 65 | 65 | 65 | 65 |
| 2 threads | 1214 | 1185 | 1186 | 1214 | 1213 |
| 3 threads | 1699 | 1183 | 1194 | 1237 | 1225 |
| 5 threads | 3694 | 2040 | 2046 | 1776 | 1710 |
| 7 threads | 4798 | 2416 | 2076 | 1979 | 2119 |
| 8 threads | 6309 | 2947 | 2457 | 2366 | 2324 |

According to this benchmark, the sweet spot is around $m = 4$ , where the barrier is not too expensive for either low or high number of threads. We will then use $m = 4$.

*Benchmarking the barriers* We implemented these four barriers with a common interface and then benchmarked them by doing 1000000 successive calls to the `wait ()` function from the given number of threads :

Table 2: Average barrier `wait ()` cost (in cycles)

| | POSIX | Sem. | Spin | Tree Spin |
|---|---|---|---|---|
| 1 thread | 782 | 18 | 90 | 65 |
| 2 threads | 56145 | 59541 | 661 | 1214 |
| 3 threads | 31536 | 42567 | 1263 | 1237 |
| 5 threads | 135797 | 66994 | 5195 | 1776 |
| 7 threads | 192047 | 82679 | 7734 | 2366 |

The benchmark thus proves that the POSIX and semaphore based barriers are way more expensive than the spinlock-based ones. While the global spinlock based barrier is more efficient with only two threads, the tree shaped spinlock barrier is the preferred choice with 3 threads and more, and has thus been chosen as the default for our simulations.

## 3.3 Efficient thread-safe reference counting

The reference counting is a key performance point, since it is used by most ns-3 objects, so that making Ref/Unref operations slower will immediately hit the overall performances of the whole simulator, *i.e.* the non multithreaded simulator implementation will also take the hit. Consequently, we need an implementation of thread-safe reference counting which is as efficient as possible in both unithreaded and multithreaded situations.

### *3.3.1 Lock-based reference counting*

The most trivial solution to make the reference counting thread-safe is simply to acquire a *mutex* (*i.e.* a mutually exclusive lock, which can only be owned by a single thread at once, so that when a mutex is already acquired, subsequent calls to the lock function block until the mutex is actually acquired) before reading or writing the reference count and release it afterwards :

```
void RefCounted::Ref (void)
{
  pthread_mutex_lock (&m_lock);
  m_count++;
  pthread_mutex_unlock (&m_lock);
}

void RefCounted::Unref (void)
{
  pthread_mutex_lock (&m_lock);
  m_count--;
  if (m_count == 0)
```

```
  {
    pthread_mutex_unlock (&m_lock);
    delete this;
  }
  else
    pthread_mutex_unlock (&m_lock);
}
```

This very simple solution sadly has a huge downside : microbenchmarks have shown that this solution is between 20 and 100 times slower than the corresponding raw operations.

### 3.3.2 Lockless reference counting

Consequently, a smarter implementation of thread-safe reference counting is required. The common way of doing it is based on platform specific atomic operations.

*Atomic operations* These operations are memory operations which are guaranteed to be applied on a consistent way even if the operations take more than one cycle to execute. For instance, reading a 64 bits integer on a 32 bits architecture could be done by reading two 32 bits integer sequentially, so that using non-atomic operations the first integer could be read while the second one is being modified so that the final 64 bits integer would neither be the original one nor the new one. An atomic read would instead protect the memory line, read both integers and release the memory.

Through this document, we will use the following operations :

- *get*, which atomically reads an integer and returns its value.
- *exchange-&-add*, which atomically adds a given value to the target integer and returns the old value.
- *compare-&-exchange*, which atomically compares the target integer with a given value, and if they are equal replace the target integer by another value. It returns the value of the target integer before the comparison.

Using these operations, we can now implement simple lockless reference counting :

```
void RefCounted::Ref (void)
{
  AtomicExchangeAndAdd (&m_count, 1);
}

void RefCounted::Unref (void)
{
  if (AtomicExchangeAndAdd (&m_count, -1) == 1)
    delete this;
}
```

The following variant of the Unref function has proved being cheaper for short lived objects. The idea is that reading first (cheap operation) may avoid useless updates (expensive operation), which works well for short-lived objects Ref/Unref'd only once or twice.

```
void RefCounted::Unref (void)
{
  int old, cur;
retry_atomic_decrement:
  old = AtomicGet (&m_count);
  if (old == 1)
  {
    cur = AtomicCompareAndExchange (&m_count,
                                    old,
                                    old - 1);
    if (cur != old)
      goto retry_atomic_decrement;
  }
  else
    delete this;
}
```

The following benchmark compares the performances of those implementations with the raw and locked ones. The benchmark concurrently runs the specified number of threads which are all Ref/Unref'ing a single counter.

Table 3: Average Ref/Unref cost (in cycles)

| | Raw | Locked | Add[1] | Compare[2] |
|---|---|---|---|---|
| 1 thread | 13 | 208 | 55 | 71 |
| 2 threads | unsafe | 1353 | 208 | 323 |
| 5 threads | unsafe | 10526 | 639 | 1559 |

[1] Atomic *exchange-&-add* based implementation

[2] Atomic *compare-&-exchange* based implementation

This benchmark clearly shows that the *exchange-&-add* is the most efficient implementation for now. Yet, there may still be room for improvement, given that the concurrent access to the reference counters are quite rare, which is a property we have not tried to exploit yet.

### 3.3.3 Doing thread-local reference counting

Consequently, a simple solution could be to add a per-thread reference counting to each object. This way, the common reference counter would only have to be increased or decreased when a thread starts or stops referencing an object (that is, when the per-thread counter becomes greater than 0 or when it reaches 0 again), which should greatly reduce the number of required atomic operations. This per-thread reference counting can be implemented either in an intrusive way, at the object level, or in a non intrusive way, through the smart pointers that handle the higher level of the reference count.

*Thread-local refcounting : the intrusive way* Doing intrusive reference counting means that the object itself holds the counter. This is the current way reference counting is implemented in ns-3. Adding a thread-local

reference counter is just a matter of associating a TLS (for Thread Local Storage) data to each object. This can be easily done using the POSIX thread-specific data implementation. Upon creation, each reference counted object is associated to a key, which acts like an index into an array of thread specific data. Each thread can then get or set the corresponding data using the key. Since the default value is 0 for keys and that, by nature, a thread has 0 reference on a given object, things are even easier to implement.

```
void RefCounted::Ref (void)
{
  long count = pthread_getspecific (m_key);
  pthread_setspecific (m_key, count - 1);
  if (count == 0)
    AtomicExchangeAndAdd (&m_count, 1);
}

void RefCounted::Unref (void)
{
  long count = pthread_getspecific (m_key);
  pthread_setspecific (m_key, count - 1);
  if (count == 1)
  {
    if (AtomicExchangeAndAdd (&m_count, -1) == 1)
      delete this;
  }
}
```

We benchmarked this implementation using the same method as before :

Table 4: Average Ref/Unref cost (in cycles)

| | Raw | Add[1] | TLS[2] |
|---|---|---|---|
| 1 thread | 13 | 55 | 44 |
| 2 threads | unsafe | 208 | 44 |
| 5 threads | unsafe | 639 | 44 |

[1] Atomic *exchange-&-add* based implementation

[2] TLS based intrusive implementation

These results show that this is the best implementation so far, with a constant time with the number of threads and a lower number of cycles per Ref/Unref pair than the atomic operations based implementations.

Sadly, this implementation, which was promising during the benchmark phase, just cannot scale, because the number of POSIX TLS keys is bounded by an upper limit, identified by the constant `PTHREAD_KEYS_MAX`, which is 1024 on most systems. This is actually easy to understand : the memory for the thread specific data is not allocated and reallocated on the fly but rather allocated once when the thread is created, so that the size of the allocated memory must be known at this point, so it has to be either computed at compile-time (which is not possible in this case, we never know how much objects will be used in the simulation) or bounded by a constant. Yet, the consequence is that a network with just 1025 nodes and nothing else (no link between nodes, no application) would be enough to overflow the reference-counting system.

To workaround this issue, we could use a single TLS key which would point to a hashmap which would hold the counts, but the hashmap lookups are likely to be too expensive to lead to any performance gain.

*Thread-local refcounting : the smart pointers way*
The other approach to doing thread-local reference counting is a non-intrusive one, which uses smart pointers. ns-3 uses its own implementation of smart pointers, known in the code as `Ptr<T>`, which automagically handles the calls to the Ref/Unref operations of the wrapped objects whenever required.

Currently, ns-3 smart pointers hold a single data : the raw pointer. The base idea of including thread-local reference counting through the smart pointers is to replace this raw pointer by a pointer to a structure holding :

- The raw pointer
- The thread identifier
- The thread-local counter

This way, the size of the `Ptr<T>` remains constant (it is the size of a raw pointer), yet holding more information. Using this, the various operators of the smart pointer are easily modified to do the thread-local reference counting. Pure Ref () calls are replaced by :

```
if (m_count->count == 0)
  {
    ((T *) m_count->ptr)->Ref ();
  }
m_count->count++;
```

Likewise, the Unref () calls are replaced by :

```
m_count->count--;
if (m_count->count == 0)
{
  ((T *) m_count->ptr)->Unref ();
  delete m_count;
}
```

And the smart pointer destruction and copy operations become :

```
template <typename T>
Ptr<T>::~Ptr ()
{
  if (m_count->ptr != 0)
  {
    m_count->count--;
    if (m_count->count == 0)
    {
      ((T *) m_count->ptr)->Unref();
      delete m_count;
    }
  }
}
```

```
template <typename T>
Ptr<T>::Ptr (Ptr const&o)
{
  uint32_t threadId = GetLocalThreadId ();
  struct LocalCount *otherCount = o.m_count;
  if (threadId == otherCount->threadId)
  {
    m_count = otherCount;
  }
  else
  {
    m_count = new struct LocalCount;
    m_count->ptr = otherCount->ptr;
    m_count->count = 0;
    m_count->threadId = threadId;
  }
  if (m_count->ptr != 0)
  {
    if (m_count->count == 0)
    {
      ((T *) m_count->ptr)->Ref ();
    }
    m_count->count++;
  }
}
```

The obvious downside of this implementation is that it moves the cost of the Ref/Unref operations to the creation and deletion of the smart pointers, pairs of operations which are just as frequent. The trick is that since most structures are not used that much concurrently, much of the smart pointers creation should be almost free.

Anyway, a simple way to reduce the number of memory allocations operations is to use a free list of local counters : instead of directly creating a new counter when required, first check a thread-local list of unused counters and pick one if the list is not empty or create a new one ; instead of directly freeing an unused counter, append it to the thread-local free list.

Due to the nature of this algorithm, which performances heavily relies on how the object is used, we benchmarked it using the test case described earlier (1.3 : 'Test case'), by simply running the benchmark 10 times and averaging the resulting runtimes. Note that the *1 thread* tests were done using the non multithreaded simulator.

Table 5: Simulator runtime (in seconds)

| | Raw | Atomic | Ptr TL[1] |
|---|---|---|---|
| 1 thread | 21.3 | 35.8 | 36.2 |
| 2 threads | unsafe | 36.7 | 38.1 |
| 5 threads | unsafe | 33.4 | 35.1 |
| 8 threads | unsafe | 34.0 | 35.2 |

[1] Thread local smart pointer based implementation

These benchmarks sadly show that this method is not a win either. This is due probably due to the short lifetime of most smart pointers, which makes the cost of the creation of the local reference counts even more expensive than the shared reference count.

#### 3.3.4 *Per-thread buffered reference counting*

The thread-local reference counting attempts proved that it was hard to optimize the performance of the reference counting using per object approaches. Thus, we could try optimizing the mechanism more globally. The idea would be to stop applying Ref/Unref operations immediately when called but store them in an operation buffer. When the buffer is full, it is pushed to a central storage and then reset. The central storage is then processed at some point of the execution of the program by a single thread at once, so that there is no need to use atomic operations for the actual Ref/Unref operations application.

The actual point at which the central storage is processed is a key factor for the success of this approach. Three possibilities emerge :

- Right after a buffer is pushed to the central storage. This is a pretty intrusive solution performance-wise, but is really easy to implement.
- In a separate semi-passive thread which would just wait for new buffers to get pushed to the central storage to wake up and process them.
- In the first thread that reaches a synchronization barrier, to use its computation power instead of wasting it waiting for the other threads.

The third option, while it could be nice performance-wise, is actually both hard to implement and completely useless for non-multithreaded simulator implementations, so that we will not implement it here.

The tricky part of this idea is the conditions at which the operations may be applied, so that we do not get into situations where we apply Unref operations which lead to the deletion of an object while there were still pending Ref operations unapplied. For that we use split the Ref/Unref buffer in two separate buffers. When one of these two buffers is full, both buffers are pushed to the central storage and are atomically assigned an unique identifier which acts as a timestamp for the buffers. Then, when the central storage is processed, the Ref buffers are immediately applied, and the identifier of the last applied Ref buffer for each thread is stored in a hashmap. Using this, deciding whether apply an Unref buffer is safe or not is simple : the identifier of the Unref buffer must be lower than the minimum identifier of the last processed Ref buffer for each thread (*i.e.* the minimum value inside the hashmap). This way, we are ensured that when we process an Unref buffer, we have processed the Ref buffers that were in use in all the threads when we pushed this Unref buffer. Thus, if a

reference count drops to 0 while processing this buffer, we are ensured that it really is not referenced anymore.

To summarize, the algorithm is the following :

```
PushLocalBuffers:
  stored_buffers->uid = next_uid++
  stored_buffers->ref_buffer = localRefBuffer
  stored_buffers->unref_buffer = localUnrefBuffer
  stored_buffers->threadId = localThreadId
  store stored_buffers into central storage
ProcessBuffers:
  for each unprocessed buffer B:
    process B->ref_buffer
    lastThreadRefUid[B->threadId] = B->uid
    push B to a queue of Unref buffers
  for each buffer B in queue of Unref buffers:
    if B->uid <= min (lastThreadRefUid):
      process B->unref_buffer
      delete B
```

Just as the previous approach, the performance of this algorithm heavily depends on the actual behavior of the simulation, so that we used the same benchmarking method to measure the performance of both immediate processing and dedicated thread processing.

Table 6: Simulator runtime (in seconds)

| | Raw | Atomic | B-I[1] | B-T[2] |
|---|---|---|---|---|
| 1 thread | 21.3 | 35.8 | 28.6 | 54.8 |
| 2 threads | unsafe | 36.7 | 30.9 | 50.5 |
| 5 threads | unsafe | 33.4 | 26.8 | 50.0 |
| 8 threads | unsafe | 34.0 | 26.5 | 48.6 |

[1] Buffered refcounting with immediate processing

[2] Buffered refcounting with dedicated thread

This approach seems to be a win, at last. The non multithreaded simulator is now only 33% slower than the non thread safe one, which is worth it, and with no specific workload balancing optimization, we get a runtime increase of only 25%.

The dedicated processing thread implementation curiously lead to a huge drop of performances. This is probably due to the use of locks to protect the central storage which induces quite a bit of contention.

The little downside of this algorithm is the memory usage. While it is still much better than without garbage collection, the use of temporary buffers increases the memory usage during the execution, especially with several threads. Adjusting the buffer size, which was set to 1000 operations in the previous benchmark, could both improve performances even more and optimize memory usage by reducing the number of created buffers, thus shortening their lifetime (since it will be easier to satisfy the Unref buffer processing condition).

*Adjusting buffer size* We ran the same benchmarks, using buffer sizes of 10000 and 100000 operations :

Table 7: Simulator runtime (in seconds)

| | I[1] 10k | T[2] 10k | I[1] 100k | T[2] 100k |
|---|---|---|---|---|
| 1 thread | 27.4 | 30.8 | 28.2 | 32.9 |
| 2 threads | 26.6 | 29.6 | 27.7 | 30.7 |
| 5 threads | 26.4 | 27.7 | 27.6 | 29.3 |
| 8 threads | 26.4 | 28.7 | 27.6 | 29.3 |

[1] Buffered refcounting with immediate processing

[2] Buffered refcounting with dedicated thread

These benchmarks show that using a buffer size of 10000 operations is the best option, since it speeds up even more the application, optimizing the buffer copy frequencies while avoiding having to handle huge buffers, which is the problem with larger orders of magnitude of buffer size. Quick memory benchmarks also show that this is the value that optimizes the average amount of memory used by the simulator.

### 3.3.5 *Dealing with reference counted aggregated objects*

ns-3 uses a mechanism of aggregation to link all the objects related to a given node, such as the applications running on it or the networking stacks. This mechanism eases the access of these objects between them (if an aggregated object wants to access another object of the aggregation cycle, it just needs to do something along the line `this->GetObject<TypeOfOtherobject> ()`) and nicely prevents the creation of cycles of reference counted objects, for which additional heuristics would be required for automatic memory handling, heuristics which may even be impossible to use without major rework of the ns-3 object system.

Currently, when one of the objects in the aggregation cycle loses its last reference, the following algorithm is run :

```
for each aggregated object O:
  if O.refcount != 0:
    return
for each aggregated object O:
  O.dispose ()
for each aggregated object O:
  delete O
```

The first `for` loop checks that all the aggregated objects are not referenced any more. The second loop calls the `dispose ()` method on each object, which is a destructor-like method which attempts to prepare the object for deletion, for instance by dropping any reference to other objects it may hold. The last loop actually deletes the objects.

An idea to simplify this algorithm and reduce the number of reference counters in use (and so the amount of data living in the caches) is to use a single counter for the whole cycle of aggregated objects. This can be easily done by replacing the current counter by a pointer to

the actual counter and sharing this pointer when doing the aggregation (the read and write operation on the counters values are done atomically) :

```
aggregate O1 to O2:
  other_counter = O1.counter
  for each object O in O1 aggregation cycle:
    O.counter = O2.counter
  O2.counter.count += other_counter.count
  merge O1 and O2 aggregation cycles
```

This way, the previous deletion algorithm is now only called when all the references on all the objects of the aggregation cycle have been dropped, so that the algorithm is called only once per aggregation cycle and is always successful.

The only downside of this algorithm is the use of shared reference counters which have to be dynamically allocated. A free list could be used in a similar manner than for the smart pointers thread-local reference counting, but this list would have to be common to all the threads, which would once again lead to some extra overhead.

We benchmarked this algorithm using the same method as before :

Table 8: Simulator runtime (in seconds)

| | Raw | Atomic | B-I | CC[1] |
|---|---|---|---|---|
| 1 thread | 21.3 | 35.8 | 27.4 | 34.2 |
| 2 threads | unsafe | 36.7 | 26.6 | 36.2 |
| 5 threads | unsafe | 33.4 | 26.4 | 31.6 |
| 8 threads | unsafe | 34.0 | 26.4 | 31.3 |

[1] Common Counter per object aggregation cycle

This last attempt at improving the reference counting performance is a failure, which is mostly due to the creation and deletion of counters when objects are created and aggregated.

Consequently, we will keep the buffered reference counting with immediate applications, at least until the contention issues induced by the use of a dedicated thread for buffers processing are solved.

## 3.4 Dealing with memory issues

### *3.4.1 Data relocation & access contention*

The solution to data relocation between threads is just to avoid it, by dedicating a thread to the processing each partition. This way, most of the data does not have to be moved from one thread to another (mostly only the related packets and events still go through threads). Likewise, it reduces the amount of contention (*i.e.* the amount of time spent waiting for a shared resource) by removing the need of a global structure from which the partitions are picked up by the threads, structure which obviously needs to be thread-safe and, as such, a point of contention.

Yet, this solution is obviously not perfect since it absolutely does not ensure that the threads workloads will be balanced : some partitions may be key points of the network (network backbone parts, for instance) and take a lot more time to process than other partitions. Thus, the threads not handling these partitions would waste a lot of time waiting at the barrier that the other threads finish processing them.

### *3.4.2 Workload balancing*

Sadly the main solution to the problem of workload balancing is exactly the opposite of the previous solution : to avoid threads wasting time waiting for each other, the workload must be balanced at runtime between threads, which basically means having a shared structure from which each thread picks up partitions to process. Still, we can make sure that the implementation of this shared structure is as optimized as possible to reduce contention.

The easiest way to implement this shared structure is simply to use a `std::queue` which access is protected by a lock. Obviously, the performance issues which emerged in 3.3.1 : 'Lock-based reference counting' also appear here. A more efficient solution is to use a `std::vector` and a shared index of the current partition, index which is locked instead of the underlying structure.

The obvious solution is thus to use atomic operations once again : instead of locking the shared index, increment it atomically using *exchange-&-add*.

Yet, there is still some contention related to the fact that all the threads share a single data : an oprofiled run of the simulator showed that 14.9% of the runtime was spent in the function which gets the next available shared partition, incrementing the index.

*Using several partitions vectors* The solution to reduce contention related to the use of a shared index is simply to split the shared partition set into several sets, each shared by a given number of threads. This is similar to the solution we used in 3.2 : 'Tree shaped spinlock based barrier'.

Yet, this is a little more complicated, because the primary use of these shared sets is to balance the workload : a similar unbalance of the workload could occur if there are two much lists. Consequently, the best number of shared partitions sets heavily depends on the number of threads.

We benchmarked this idea using our test case in order to find out which number of shared sets is the best :

Table 9: Simulator runtime (in seconds)

| Shared sets | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 2 threads | 26.6 | 27.8 | N/A | N/A |
| 5 threads | 26.4 | 25.8 | 25.9 | 27.0 |
| 8 threads | 26.4 | 25.3 | 24.7 | 25.9 |

This benchmark shows that the best number of shared sets is between a half and a third the number of threads. For instance, 3 shared sets seems to be the most efficient for 8 threads, while with 5 threads 2 sets are more efficient than 3.

#### *3.4.3 Balancing the solutions*

Now that we have solutions for each problem, we need to balance them, *i.e.* find the right proportion of the partitions which should be dedicated to each thread. The threads will then first process partitions from their dedicated list, which will avoid much of the contention on the shared data sets, and once they finished processing this list, they will start processing partitions from the shared sets, until these sets are empty. The proportion of dedicated partitions is definitely the key here : if it is too low, the processes will quickly finish processing their partition list and the contention problems will reappear ; if it is too high, there will not be enough work to do in the shared sets to balance the workload.

The benchmarks were run on an octocore machine, with 2 CPUs with each 4 cores (2 cores on 2 split dies, with 6MB of CPU cache on each die) and 16GB of RAM.

We benchmarked our test case with various percentages of dedicated partitions to find the most appropriate. We used, for each number of threads, the best number of shared sets as found in the previous subsection :

Table 10: Simulator runtime (in seconds)

| Dedicated % | 0 | 25 | 50 | 75 | 100 |
|---|---|---|---|---|---|
| 2 threads | 26.6 | 26.4 | 26.2 | 26.7 | 28.5 |
| 5 threads | 25.8 | 25.3 | 23.4 | 25.4 | 28.2 |
| 8 threads | 24.7 | 23.6 | **22.1** | 22.8 | 26.1 |

This benchmark tends to show that our test case behaved the best with 50% of thread-dedicated partitions. Even better, for the best case, *i.e.* 8 threads, 3 shared sets and 50% dedicated partitions, the simulator was only 3% slower than the original non thread-safe one. This may seem ironical, since our whole goal was speeding up the simulator, but it is already a great achievement after the slowdown brought by thread-safety. The best thing is that the actual simulation time (*i.e.* excluding the initialization phase) is actually much better than before : it was initially of 11.5 seconds, and is, with this best case, of only 9.1 seconds, which is a 20% speedup !

## 4 FURTHER WORK

### 4.1 Better test cases

One of the problems we encountered while working on this project was the lack of appropriate test cases. The tests we were able to gather were most often tests for other features, such as routing, which lead to very specific edge cases for the multithreading stuff, with for instances some events taking much more time than most others (*e.g.* 1000 times slower) which lead to an heavy unbalance of the workload over the threads. Finding better tests cases and more specifically classic parallel network simulation situations is required for deeper understanding of the simulator behavior.

### 4.2 Lockless buffered reference counting

The buffered reference counting, described in 3.3.4 : 'Per-thread buffered reference counting', was a great performance improvement, but it probably could be even better if the buffer processing was done in a separate thread, so that the reference counting would be even lighter for the simulator performances. For now, this approach is not efficient because the buffer handling operations are using lock based structures, which lead to heavy contention points (and to extra CPU cache flushes). Thus, re-engineering the buffer handling to be lockless, or at least thoroughly investigating its behavior to understand and fix or lighten contention points, would probably lead to new performance gains for the reference counting.

### 4.3 Even smarter load balancing

While the dedicated/shared partitions sets split helped improving load balancing a lot, there is still room for improvement. Other ideas could be put in practice quite easily to improve workload balancing even more. For instance, one of the ideas we experimented (but which is not included in the current patchset) is to sometimes dynamically relocate a dedicated partition from a thread to another one, based on the thread performance at a given iteration (we would move a partition from the slowest thread to the fastest one). This way, we could balance the dedicated partitions sets over time. Such heuristics could significantly improve performance and should thus be investigated.

### 4.4 Lookahead for wireless networks

Computing the lookahead for wireless networks is not a trivial job. The transmission delay heavily depends on the peers spacial situation, and trivially computing the

minimum delay between all peers would be a $\Theta(n^2)$ operation (with $n$ the number of wireless stations) at each simulator iteration, which would be really expensive. Furthermore, approximating the minimum delay by 0 might not either be a good solution performance-wise.

Consequently sorting out the correct solution for computing this lookahead will require some more careful thinking.

### 4.5 Efficient packet-related caches

As mentioned in 2.3 : 'Transparent thread-safety', packet data and metadata buffers and caches have been disabled because of possible thread-safety problems. Finding out the best solution for each of these caches, such as keeping them disabled, replacing them with thread-local equivalents or protecting them using lockless thread-safe structures, is a must-have for the multithreaded simulator to be ready for the prime-time.

## CONCLUSION

Doing multithreaded simulation is not an easy job. Doing it seamlessly is a really hard job. The initial cost of transparently making an event simulator thread safe is much bigger than one could expect. The additional cost for having to do the workload balancing automatically, without any hint on how the network should be partitioned, is also huge. Yet, with a good amount of smart algorithms and thorough analysis of each aspect of the behavior of the simulator, we eventually improved the actual simulation performance by 20%, and there is still room for improvement, with the paths we were not able to follow during this internship. The user-friendliness goal is also reached, since all the user will ever need to enable the multithreaded simulator for his simulation is adding two lines of code, one specifying that the multithreaded implementation should be used, the other one running a function which sets up everything correctly (creating the partitions...).

Working on this simulator implementation also meant discovering and practicing a strong research method, which can be summarized by the following : understand the problem, quantify it, use the right tools, get a fresh look. When a problem appeared after a code change, the first thing was to understand why it happened, then quantify how much it was hitting the simulator. This can only be achieved by using the right tools : debuggers, profilers, manual instrumentation... And the last point is the greatest : if anything goes wrong and you cannot figure it out, bring one of your coworkers on the problem. Team work is the key.

## ACKNOWLEDGMENTS

Huge thanks to Mathieu Lacage and Martín Ferrari for these two months and a half of great computer science, software engineering and geekery.

## References

[Bryant, 1977] Bryant, R. E. (1977). Simulation of packet communication architecture computer systems. Technical report, Cambridge, MA, USA. `http://portal.acm.org/citation.cfm?id=889797`.

[Chandy and Misra, 1979] Chandy, K. M. and Misra, J. (1979). Distributed simulation: A case study in design and verification of distributed programs. *Software Engineering, IEEE Transactions on*, SE-5(5):440–452. `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1702653`.

[Fujimoto, 2000] Fujimoto, R. M. (2000). *Parallel and Distributed Simulation Systems*. Wiley Interscience.

[Nicol, 2003] Nicol, D. (2003). Darpa network modeling and simulation (nms) baseline network topology. `http://www.ssfnet.org/Exchange/gallery/baseline/index.html`.