A Tale Of Lock-Free Agents

Towards Software Transactional Memory in parallel Agent-Based Simulation

JONATHAN THALER and PEER-OLAF SIEBERS, University of Nottingham, United Kingdom

With the decline of Moore's law and the ever increasing availability of cheap massively parallel hardware, it becomes more and more important to embrace parallel programming methods to implement Agent-Based Simulations (ABS). This has been acknowledge in the field a while ago and numerous research on distributed parallel ABS exists, focusing primarily on Parallel Discrete Event Simulation as the underlying mechanism. However, these concepts and tools are inherently difficult to master and apply and often overkill in case implementers simply want to parallelise their own, custom agent-based model implementation. However, with the established programming languages in the field, Python, Java and C++, it is not easy to address the complexities of parallel programming due to unrestricted side effects and the intricacies of low-level locking semantics. Therefore, in this paper we propose the use of a lock-free approach to parallel ABS using Software Transactional Memory (STM) in conjunction with the pure functional programming language Haskell, which in combination, removes some of the problems and complexities of parallel implementations in imperative approaches.

We present two case studies where we compare the performance of lock-based and lock-free STM implementations in two different well known Agent-Based Models, where we investigate both the scaling performance under increasing number of CPUs and the scaling performance under increasing number of agents. We show that the lock-free STM implementations consistently outperform the lock-based ones and scale much better to increasing number of CPUs both on local machines and on Amazon Cloud Services. Further by utilizing the pure functional language Haskell we gain additional benefits like immutable data and lack of side effects guaranteed at compile-time, something of fundamental importance and benefit in parallel programming in general and scientific computing like ABS in particular.

CCS Concepts: • **Computing methodologies** → **Agent / discrete models**; **Massively parallel and high-performance simulations**; *Simulation languages*;

Additional Key Words and Phrases: Agent-Based Simulation, Software Transactional Memory, Parallel Programming, Concurrency, Functional Programming, Haskell

ACM Reference Format:
Jonathan Thaler and Peer-Olaf Siebers. 2018. A Tale Of Lock-Free Agents: Towards Software Transactional Memory in parallel Agent-Based Simulation. *ACM Trans. Model. Comput. Simul.* 28, 4, Article 0 (October 2018), 25 pages. https://doi.org/0000001.0000001

Authors' address: Jonathan Thaler, jonathan.thaler@nottingham.ac.uk; Peer-Olaf Siebers, peer-olaf.siebers@nottingham.ac.uk, University of Nottingham, 7301 Wollaton Rd, Nottingham, NG8 1BB, United Kingdom.

XXXX-XXXX/2018/10-ART0 $15.00
https://doi.org/0000001.0000001

1 INTRODUCTION

The future of scientific computing in general and Agent-Based Simulation (ABS) in particular is parallelism: Moore's law is declining as we are reaching the physical limits of CPU clocks. The only option is going massively parallel due to availability of cheap massive parallel local hardware with many cores, or cloud services like Amazon EC. This trend has been already recognised in the field of ABS as a research challenge for *Large-scale ABMS* [24] was called out and as a substantial body of research for parallel ABS shows [1, 4, 9, 10, 13, 15, 20, 21, 27, 31, 33, 34].

In this body of work it has been established that parallelisation of autonomous agents, situated in some spacial, metric environment can be particularly challenging. The reason for this is that the environment constitutes a key medium for the agents interactions, represented as a *passive* data structure, recording attributes of the environment and the agents [20]. Thus the problem of parallelising ABM boils down to the problem of how to synchronise access to shared state without violating the causality principle and resource constraints [21, 34]. Various researchers have developed different techniques but they are all ultimately based on the concept of Parallel Discrete-Event Simulation (PDES). The idea behind PDES is to partition the shared space into logical processes which run at their own speed, processing events coming from themselves and other logical processes. To deal with inconsistencies there exists a conservative approach which does not allow to process events with a lower timestamp than the current time of the logical process; and an optimistic approach which deals with inconsistencies through rolling back changes to state.

Adopting PDES to ABM is challenging as agents are autonomous and thus the topology can change in every step, making it hard to predict the topology of logical processes in advance [20] and thus posing a difficult problem for parallelisation in general [4]. The work [33, 34] discusses this challenge by giving a detailed and in-depth discussion of the internals and implementation of their powerful and highly complex PDES-MAS system. The rather conceptual work [21] proposes a general, distributed simulation framework for multiagent systems and addresses a number of key problems: decomposition of the environment, load balancing, modelling, communication and shared state variables, which the authors mention as the central problem of parallelisation.

In addition, various distributed simulation environments for ABM have been developed and their internals published in research papers: the SPADES system [31] manages agents through UNIX pipes through a parallel sense-think-act cycle employing a conservative PDES approach; Mace3J [9] a Java based system running on single- or multicore workstations implementing a message passing approach to parallelism, James II [15] also a Java based system with focus on PDEVS simulation with a plugin architecture to facilitate reuse of models, RePast-HPC [10, 27] using a PDES engine under the hood.

The baseline of this body of research is that parallelisation is possible and we know how to do it. However, the complexity of these parallel and distributed simulation concepts and toolkits is high and the model development effort is hard [1]. Further, this sophisticated and powerful machinery is not always required as ABM does not always need to be run in a distributed way but the implementers 'simply' want to parallelise their models locally. Although these existing distributed ABM frameworks could be used for this, they are overkill and more straightforward concepts for parallelising ABM would be more appropriate. However, for this case there does not exist much research, and it seems that either implementers resorted to the distributed ABM frameworks, implemented their own low-level concurrency plumbing which can be considerably complex - or simply refrained from using parallelism due

to the high complexity involved and accept a longer execution time. What makes it worse is that parallelism always comes with danger of additional, very subtle bugs, which might lie dormant, potentially invalidating significant scientific results of the model. Therefore something simpler is needed for local parallelism. Unfortunately, the established imperative languages in the ABS field, Python, Java, C++, follow mostly a lock-based approach to concurrency which is error prone and does not compose. Further, data-parallelism in an imperative language is susceptible to side-effects because these languages cannot distinguish between data-parallelism and concurrency in the types.

Therefore, this paper proposes Software Transactional Memory (STM) in conjunction with functional programming in Haskell [17] as a new underlying concept for local parallelisation of ABM. The paper [5] gives a good indication how difficult and complex constructing a correct concurrent program is and shows how much easier, concise and less error-prone an STM implementation is over traditional locking with mutexes and semaphores. Further it shows that STM consistently outperforms the lock-based implementation. We follow this work and compare the performance of lock-based and STM implementations and hypothesise that the reduced complexity and increased performance will be directly applicable to ABS as well.

Therefore, this paper hypothesises that by using STM in Haskell, implementing local parallel ABM is considerably easier than with lock-based approaches, less error prone and easier to validate. Although STM exists in other languages as well, Haskell was one of the first to natively build it into its core. Further, it has the unique benefit that it can guarantee the lack of persistent side effects at compile time, allowing unproblematic retries of transactions, something of fundamental importance in STM. This makes the use of STM in Haskell very compelling.

To the best of our knowledge we are the first to *systematically* discuss the use of STM in the context of ABM. However, the idea of applying transactional memory to simulation in general is not new and was already explored in the work [13], where the authors looked into how to apply Intel's *hardware* transactional memory to simulations in the context of a Time Warp PDES simulation. The results showed that their approach generally outperformed traditional locking mechanisms.

In his master thesis [3] the author investigates Haskells' parallel and concurrency features to implement (amongst others) *HLogo*, a Haskell clone of the NetLogo [37] simulation package, focusing on using STM for a limited form of agent-interactions. *HLogo* is basically a re-implementation of NetLogos API in Haskell where agents run within an unrestricted context (known as `IO`) and thus can also make use of STM functionality. The benchmarks show that this approach does indeed result in a speed-up especially under larger agent-populations. The authors' thesis can be seen as one of the first works on ABS using Haskell. Despite the concurrency and parallel aspect our work share, our approach is rather different: we avoid IO within the agents under all costs and explore the use of STM more on a conceptual level rather than implementing a ABS library and compare our case-studies with lock-based and imperative implementations.

We present two case-studies in which we employ an agent-based spatial SIR [23, 35] and the well known SugarScape [7] model to test our hypothesis. The latter model can be seen as one of the most influential exploratory models in ABS which laid the foundations of object-oriented implementation of agent-based models. The former one is an easy-to-understand explanatory model which has the advantage that it has an analytical theory behind it which can be used for verification and validation.

The aim of this paper is to experimentally investigate the benefit of using STM over lock-based approaches for concurrent ABS models. The contribution of this paper is a systematic investigation of the usefulness of STM over lock-based approaches, therefore giving implementers a new method of locally parallelising their own implementations without the overhead of a distributed, parallel PDES sytsem or the error-prone low-level locking semantics of a custom built parallel implementation. Therefore, our paper directly addresses the *Large-scale ABMS* challenge [24], which focuses on efficient modelling and simulating large-scale ABS. Further, using STM, which restricts side effects, and makes concurrency easier, can help in the validation challenge [24] *H5: Requirement that all models be completely validated*.

We start with Section 2 where we discuss the concepts of STM and Haskell. In Section 3 we show how to apply STM to ABS in general. Section 4 contains the first case-study using a spatial SIR mode whereas Section 5 presents the second case-study using the SugarScape model. We conclude in Section 6 and give further research directions in Section 7.

2 BACKGROUND

2.1 Software Transactional Memory

Software Transactional Memory (STM) was introduced by [32] in 1995 as an alternative to lock-based synchronisation in concurrent programming which, in general, is notoriously difficult to get right. This is because reasoning about the interactions of multiple concurrently running threads and low level operational details of synchronisation primitives is *very hard*. The main problems are:

- Race conditions due to forgotten locks;
- Deadlocks resulting from inconsistent lock ordering;
- Corruption caused by uncaught exceptions;
- Lost wake-ups induced by omitted notifications.

Worse, concurrency does not compose. It is very difficult to write two functions (or methods in an object) acting on concurrent data which can be composed into a larger concurrent behaviour. The reason for it is that one has to know about internal details of locking, which breaks encapsulation and makes composition dependent on knowledge about their implementation. Therefore, it is impossible to compose two functions e.g. where one withdraws some amount of money from an account and the other deposits this amount of money into a different account: one ends up with a temporary state where the money is in none of either accounts, creating an inconsistency - a potential source for errors because threads can be rescheduled at any time.

STM promises to solve all these problems for a low cost by executing actions *atomically*, where modifications made in such an action are invisible to other threads and changes by other threads are invisible as well until actions are committed - STM actions are atomic and isolated. When an STM action exits, either one of two outcomes happen: if no other thread has modified the same data as the thread running the STM action, then the modifications performed by the action will be committed and become visible to the other threads. If other threads have modified the data then the modifications will be discarded, the action block rolled-back and automatically restarted.

STM in Haskell is implemented using optimistic synchronisation, which means that instead of locking access to shared data, each thread keeps a transaction log for each read and write to shared data it makes. When the transaction exits, the thread checks whether it has a

consistent view to the shared data or not: whether other threads have written to memory it has read.

In the paper [14] the authors use a model of STM to simulate optimistic and pessimistic STM behaviour under various scenarios using the AnyLogic simulation package. They conclude that optimistic STM may lead to 25% less retries of transactions. The authors of [30] analyse several Haskell STM programs with respect to their transactional behaviour. They identified the roll-back rate as one of the key metric which determines the scalability of an application. Although STM might promise better performance, they also warn of the overhead it introduces which could be quite substantial in particular for programs which do not perform much work inside transactions as their commit overhead appears to be high.

2.2 Parallelism, Concurrency and STM in Haskell

In our case-study implementations we are using the functional programming language Haskell. The paper of [17] gives a comprehensive overview over the history of the language, how it developed and its features and is very interesting to read and get accustomed to the background of the language. Note that Haskell is a *lazy* language which means that expressions are only evaluated when they are actually needed.

2.2.1 Side-Effects. One of the fundamental strengths of Haskell is its way of dealing with side-effects in functions. A function with side-effects has observable interactions with some state outside of its explicit scope. This means that the behaviour depends on history and that it loses its referential transparency character, which makes understanding and debugging much harder. Examples for side-effects are (amongst others): modify a global variable, await an input from the keyboard, read or write to a file, open a connection to a server, drawing random-numbers, etc.

The unique feature of Haskell is that it allows to indicate in the *type* of a function that it does have side-effects and what kind of effects they are. There are a broad range of different effect types available, to restrict the possible effects a function can have e.g. drawing random-numbers, sharing read/write state between functions, etc. Depending on the type, only specific operations are available, which is then checked by the compiler. This means that a program in which one tries to e.g. read a file in a function which only allows drawing random-numbers will fail to compile.

Here we are only concerned with two effect types: The *IO* effect context can be seen as completely unrestricted as the main entry point of each Haskell program runs in the *IO* context which means that this is the most general and powerful one. It allows all kind of I/O related side-effects: reading/writing a file, creating threads, write to the standard output, read from the keyboard, opening network-connections, mutable references, etc. Also the *IO* context provides functionality for concurrent locks and global shared references. The other effect context we are concerned with is *STM* and indicates the STM context of a function - we discuss it below.

A function without any side-effect type is called *pure*. A function with a given effect-type needs to be executed with a given effect-runner which takes all necessary parameters depending on the effect and runs a given effectful function returning its return value and depending on the effect also an effect-related result. Note that we cannot call functions of different effect-types from a function with another effect-type, which would violate the guarantees. Calling a *pure* function though is always allowed because it has, by definition, no side-effects.

Although such a type-system might seem very restrictive at first, we get a number of benefits from making the type of effects we can use explicit. First we can restrict the side-effects a function can have to a very specific type which is guaranteed at compile time. This means we can have much stronger guarantees about our program and the absence of potential run-time errors. Second, by the use of effect-runners, we can execute effectful functions in a very controlled way, by making the effect-context explicit in the parameters to the effect-runner.

2.2.2 Parallelism & Concurrency. Haskell makes a very clear distinction between parallelism and concurrency. Parallelism is always deterministic and thus pure without side-effects because although parallel code runs concurrently, it does by definition not interact with data of other threads. This can be indicated through types: we can run pure functions in parallel because for them it doesn't matter in which order they are executed, the result will always be the same due to the concept of referential transparency.

Concurrency is potentially non-deterministic because of non-deterministic interactions of concurrently running threads through shared data. Although data in functional programming is immutable, Haskell provides primitives which allow to share immutable data between threads. Accessing these primitives is only possible from within an *IO* or *STM* context (see below) which means that when we are using concurrency in our program, the types of our functions change from pure to either a *IO* or *STM* effect context.

Note that spawning tens of thousands or even millions of threads in Haskell is no problem, because threads in Haskell have a *very* low memory footprint due to being lightweight user-space threads, also known as green threads, managed by the Haskell Runtime System, which maps them to physical operating-system worker threads [25].

2.2.3 STM. The work of [11, 12] added STM to Haskell which was one of the first programming languages to incorporate STM into its main core and added the ability to composable operations. There exist various implementations of STM in other languages as well (Python, Java, C#, C/C++, etc) but we argue, that it is in Haskell with its type-system and the way how side-effects are treated where it truly shines.

In the Haskell implementation, STM actions run within the *STM* context. This restricts the operations to only STM primitives as shown below, which allows to enforce that STM actions are always repeatable without persistent side-effects because such persistent side-effects (e.g. writing to a file, launching a missile) are not possible in an *STM* context. This is also the fundamental difference to *IO*, where all bets are off because *everything* is possible as there are basically no restrictions because *IO* can run everything.

Thus the ability to *restart* a block of actions without any visible effects is only possible due to the nature of Haskells type-system: by restricting the effects to STM only, ensures that no uncontrolled effects, which cannot be rolled-back, occur.

STM comes with a number of primitives to share transactional data. Amongst others the most important ones are:

- *TVar* - A transactional variable which can be read and written arbitrarily;
- *TArray* - A transactional array where each cell is an individual shared data, allowing much finer-grained transactions instead of e.g. having the whole array in a *TVar*;
- *TChan* - A transactional channel, representing an unbounded FIFO channel;
- *TMVar* - A transactional *synchronising* variable which is either empty of full. To read from an empty or write to a full *TMVar* will cause the current thread to retry its transaction.

2.2.4 An example. We provide a short example to demonstrate the use of STM. To make it more interesting and to show the retry semantics, we use it within a `StateT` transformer where `STM` is the innermost Monad. It is important to understand that `STM` does not provide a transformer instance for very good reasons. If it would provide a transformer then we could make `IO` the innermost Monad and perform `IO` actions within `STM`. This would violate the retry semantics as in case of a retry, `STM` is unable to undo the effects of `IO` actions in general. This stems from the fact, that the `IO` type is simply too powerful and we cannot distinguish between different kinds of `IO` actions in the type, be it simply reading from a file or actually launching a missile. Lets look at the example code:

```
stmAction :: TVar Int -> StateT Int STM Int
stmAction v = do
  -- print a debug output and increment the value in StateT
  Debug.trace "increment!" (modify (+1))
  -- read from the TVar
  n <- lift (readTVar v)
  -- await a condition: content of the TVar >= 42
  if n < 42
    -- condition not met: retry
    then lift retry
    -- condition met: return content ot TVar
    else return n
```

In this example, the `STM` is the innermost Monad in a stack with a `StateT` transformer. When `stmAction` is run, it prints an `'increment!'` debug message to the console and increments the value in the `StateT` transformer. Then it awaits a condition for as long as `TVar` is less then 42 the action will retry whenever it is run. If the condition is met, it will return the content of the `TVar`. We see the combined effects of using the transformer stack were we have both the `StateT` and the `STM` effects available. The question is how this code behaves if we actually run it. To do this we need to spawn a thread:

```
stmThread :: TVar Int -> IO ()
stmThread v = do
  -- the initial state of the StateT transformer
  let s = 0
  -- run the state transformer with initial value of s (0)
  let ret = runStateT (stmAction v) s
  -- atomically run the STM block
  (a, s') <- atomically ret
  -- print final result
  putStrLn("final StateT state     = " ++ show s' ++
           ", STM computation result = " ++ show a)
```

The thread simply runs the `StateT` transformer layer with the initial value of 0 and then the `STM` computation through `atomically` and prints the result to the console. The value of `a` is the result of `stmAction` and `s'` is the final state of the `StateT` computation. To actually run this example we need the main thread to updated the `TVar` until the condition is met within `stmAction`:

```
main :: IO ()
main = do
  -- create a new TVar with initial value of 0
  v <- newTVarIO 0
  -- start the stmThread and pass the TVar
  forkIO (stmThread v)
  -- do 42 times...
  forM_ [1..42] (\i -> do
```

```
    -- use delay to 'make sure' that a retry is happening for ever increment
    threadDelay 10000
    -- write new value to TVar using atomically
    atomically (writeTVar v i))
```

If we run this program, we will see `'increment!'` printed 43 times, followed by `'final StateT state = 1, STM computation result = 42'`. This clearly demonstrates the retry semantics where `stmAction` is retried 42 times and thus prints `'increment!'` 43 times to the console. The `StateT` computation however is carried out only once and is always rolled back when a retry is happening. The rollback is easily possible in pure functional programming due to persistent data structure by simply throwing away the new value and retrying with the original value. This example also demonstrates that any `IO` actions which happen within an `STM` action are persistent and can obviously not be rolled back. `Debug.trace` is an `IO` action masked as pure using `unsafePerformIO`.

3 STM IN ABS

In this section we give a short overview of how we apply STM in our ABS. We fundamentally follow a time-driven approach in both case-studies where the simulation is advanced by some given Δt and in each step all agents are executed. To employ parallelism, each agent runs within its own thread and agents are executed in lock-step, synchronising between each Δt which is controlled by the main thread. This way of stepping the simulation is introduced in [36] on a conceptual level, where the authors name it *concurrent update-strategy*. See Figure 1 for a visualisation of our concurrent, time-driven lock-step approach.

An agent thread will block until the main thread sends the next Δt and runs the STM action atomically with the given Δt. When the STM action has been committed, the thread will send the output of the agent action to the main-thread to signal it has finished. The main thread awaits the results of all agents to collect them for output of the current step e.g. visualisation or writing to a file.

As will be described in subsequent sections, central to both case-studies is an environment which is shared between the agents using a *TVar* or *TArray* primitive through which the agents communicate concurrently with each other. To get the environment in each step for visualisation purposes, the main thread can access the *TVar* and *TArray* as well.

3.1 Adding STM to agents

We briefly discuss how to add STM to agents on a technical level and also show how to run them within their own threads. We use the SIR implementation as example - applying it to the Sugarscape implementation works exactly the same way and is left as a trivial exercise to the reader.

The first step is to simply add the `STM` Monad as the innermost level to the already the existing Transformer stack. Further, the environment is now passed as a transactional data primitive to the agent at *construction time*. Thus, the agent does not receive the `SIREnv` as input any more but receives it through currying when constructing its initial `MSF`. Further, the agent modifies the `SIREnv` directly through the `TVar`, as demonstrated in the case of the infected agent.

```
-- Make Rand a transformer to be able to add STM as innermost monad
type SIRMonad g = RandT g STM
-- Input to agent is now an empty tuple instead of the Environment
type SIRAgent g = SF (SIRMonad g) () SIRState

-- The MSF construction function takes now the TVar with the environment.
```

Fig. 1. Diagram of the parallel time-driven lock-step approach.

```
sirAgent :: RandomGen g => TVar SIREnv -> Disc2dCoord -> SIRState -> SIRAgent g

-- The infected agent behaviour is nearly the same except that
-- the agent modifies the environment through the TVar
infected :: RandomGen g => SF (SIRMonad g) () (SIRState, Event ())
infected = proc _ -> do
  recovered <- occasionally illnessDuration () -< ()
  if isEvent recovered
    then (do
      -- update the environment through the TVar
      arrM_ (lift $ lift $ modifyTVar env (changeCell coord Recovered)) -< ()
      returnA -< (Recovered, Event ()))
    else returnA -< (Infected, NoEvent)
```

The agent thread is straightforward. It takes `MVar` synchronisation primitives to synchronise with the main thread and simply runs the agent behaviour each time it receives the next *DTime*:

```
agentThread :: RandomGen g
            => Int              -- ^ Number of steps to compute
            -> SIRAgent g       -- ^ Agent behaviour MSF
            -> g                -- ^ Random-number generator of the agent
            -> MVar SIRState    -- ^ Synchronisation back to main thread
            -> MVar DTime       -- ^ Receiving DTime for next step
            -> IO ()
agentThread 0 _ _ _ _ = return () -- all steps computed, terminate thread
agentThread n sf rng retVar dtVar = do
  -- wait for dt to compute current step
  dt <- takeMVar dtVar

  -- compute output of current step
  let sfReader = unMSF sf ()
      sfRand   = runReaderT sfReader dt
```

```
      sfSTM    = runRandT sfRand rng
  -- run the STM action atomically within IO
  ((ret, sf'), rng') <- atomically sfSTM

  -- post result to main thread
  putMVar retVar ret

  -- tail recursion to next step
  agentThread (n - 1) sf' rng retVar dtVar
```

Computing a simulation step is now trivial within the main thread. All agent threads `MVars` are signalled to unblock followed by an immediate block on the `MVars` into which the agent threads post back their result. The state of the current step is then extracted from the environment, which is stored within the `TVar` which the agent threads have updated.

```
simulationStep :: TVar SIREnv      -- ^ environment
               -> [MVar DTime]     -- ^ sync dt to threads
               -> [MVar SIRState] -- ^ sync output from threads
               -> DTime            -- ^ time delta
               -> IO SIREnv
simulationStep env dtVars retVars dt = do
  -- tell all threads to continue with the corresponding DTime
  mapM_ (`putMVar` dt) dtVars
  -- wait for results but ignore them, SIREnv contains all states
  mapM_ takeMVar retVars
  -- return state of environment when step has finished
  readTVarIO env
```

4 CASE STUDY 1: SPATIAL SIR MODEL

Our first case study is the SIR model which is a very well studied and understood compartment model from epidemiology [18] which allows to simulate the dynamics of an infectious disease like influenza, tuberculosis, chicken pox, rubella and measles spreading through a population [6].

In it, people in a population of size N can be in either one of three states *Susceptible*, *Infected* or *Recovered* at a particular time, where it is assumed that initially there is at least one infected person in the population. People interact *on average* with a given rate of β other people per time-unit and become infected with a given probability γ when interacting with an infected person. When infected, a person recovers *on average* after δ time-units and is then immune to further infections. An interaction between infected persons does not lead to re-infection, thus these interactions are ignored in this model.

We followed in our agent-based implementation of the SIR model the work [23] but extended it by placing the agents on a discrete 2D grid using a Moore (8 surrounding cells) neighbourhood [**?**]. A visualisation can be seen in Figure 2.

It is important to note that due to the continuous-time nature of the SIR model, our implementation follows the time-driven [26] approach. This requires us to sample the system with very small Δt, which means that we have comparatively few writes to the shared environment which will become important when discussing the performance results.

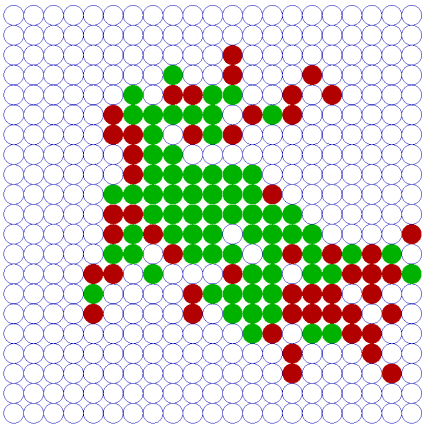


Fig. 2. Simulating the spatial SIR model with a Moore neighbourhood, a single infected agent at the center, contact rate $\beta = \frac{1}{5}$, infection probability $\gamma = 0.05$ and illness duration $\delta = 15$. Infected agents are indicated by red circles, recovered agents by green ones. The susceptible agents are rendered as blue hollow circles for better contrast.

4.1 Experiment Design

In this case study we compare the performance of the following implementations under varying numbers of CPU cores and agent numbers [1]:

(1) Sequential - This is the original implementation as discussed in [**?**] where the discrete 2D grid is shared amongst all agents as read-only data and the agents are executed sequentially within the main thread without any concurrency.
(2) STM - This is the same implementation as the *Sequential* but agents run now in the STM context and have access to the discrete 2D grid through a transactional variable *TVar*. This means that the agents now communicate indirectly by reads and writes through the *TVar*.
(3) Lock-Based - This is exactly the same implementation as the *STM* one, but instead of running in the STM context, the agents now run in IO context. They share the discrete 2D grid using a global reference and have access to a lock to synchronise access to it.
(4) RePast - To have an idea where the functional implementation is performance-wise compared to the established object-oriented methods, we implemented a Java version of the SIR model using RePast [29] with the state-chart feature. This implementation cannot run on multiple cores concurrently but gives a good estimate of the single core performance of imperative approaches. Although there exists a RePast High Performance Computing library for implementing large-scale distributed simulations in C++, we leave this for further research as an implementation and comparison is out of the scope of this paper.

Each experiment was run until $t = 100$ and stepped using $\Delta t = 0.1$ except in RePast for which we don't have access to the underlying implementation of the state-chart and left it as it is. For each experiment we conducted 8 runs on our machine (see Table 1) under no additional work-load and report the mean. Further, we checked the visual outputs and the dynamics and they look qualitatively the same as the reference *Sequential* implementation

[1]The code is freely available at https://github.com/thalerjonathan/phd/tree/master/public/stmabs/code/SIR

OS	Fedora 28 64-bit
RAM	16 GByte
CPU	Intel Quad Core i5-4670K @ 3.4GHz (4th Gen.)
HD	250Gbyte SSD
Haskell	GHC 8.2.2
Java	OpenJDK 1.8.0
RePast	2.5.0.a

Table 1. Machine and Software Specs for all experiments

	Cores	Duration
Sequential	1	72.5
Lock-Based	1	60.6
	2	42.8
	3	38.6
	4	41.6
STM	1	53.2
	2	27.8
	3	21.8
	4	20.8
RePast	1	**10.8**

Table 2. Experiments on 51x51 (2,601 agents) grid with varying number of cores.

[**?**] - we could have used more rigour and properly validated the implementations against the formal specification using tests but this was beyond the scope of this paper. In the experiments we varied the number of agents (grid size) as well as the number of cores when running concurrently (through run-time system arguments) - the numbers are always indicated clearly.

4.2 Constant Grid Size, Varying Cores

In this experiment we held the grid size constant to 51 x 51 (2,601 agents) and varied the cores where possible. The results are reported in Table 2.

Comparing the performance and scaling to multiple cores of the *STM* and *Lock-Based* implementations shows that the *STM* implementation significantly outperforms the *Lock-Based* one and scales better to multiple cores. The *Lock-Based* implementation performs best with 3 cores and shows slightly worse performance on 4 cores as can be s een in Figure 3. This is no surprise because the more cores are running at the same time, the more contention for the lock, thus the more likely synchronisation happening, resulting in higher potential for reduced performance. This is not an issue in *STM* because no locks are taken in advance.

What comes a bit as a surprise is, that the single core RePast implementation significantly outperforms *all* other implementations, even when they run on multiple cores and even with RePast doing complex visualisation in addition (something the functional implementations don't do). When looking at benchmarks [2] comparing Haskell to Java, C and C++, Haskell is significantly outperformed in most of the cases. This is a strong indication that Haskell is

[2] https://benchmarksgame-team.pages.debian.net/benchmarksgame/faster/haskell.html

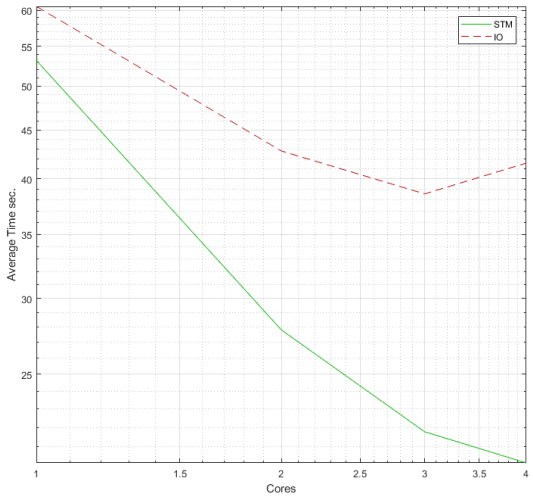


Fig. 3. Comparison of performance and scaling on multiple cores of STM vs. Lock-Based. Note that the Lock-Based implementation performs worse on 4 than on 3 cores due to lock-contention.

Grid-Size	STM	Lock-Based (4 cores)	Lock-Based (3 cores)	RePast (1 core)
51 x 51 (2,601)	20.2	41.9	38.6	**10.8**
101 x 101 (1,0201)	**74.5**	170.5	171.6	107.40
151 x 151 (22,801)	**168.5**	376.9	404.1	464.1
201 x 201 (40,401)	**302.4**	672.0	720.6	1,227.7
251 x 251 (63,001)	**495.7**	1,027.3	1,117.2	3,283.6

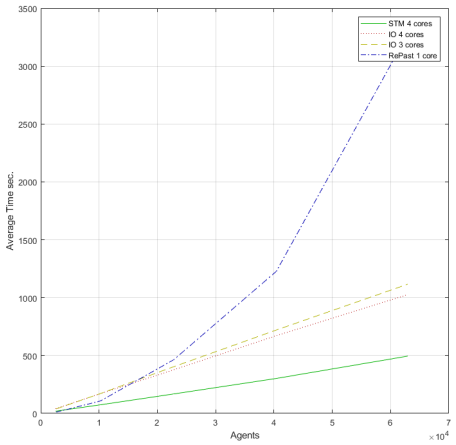
Table 3. Performance on varying grid sizes.

in general not as fast as Java, C or C++. Also, we build on the concepts developed in [**?**] to implement our ABS which make heavy use of Functional Reactive Programming, which can be substantially slower than imperative approaches [16, 28].

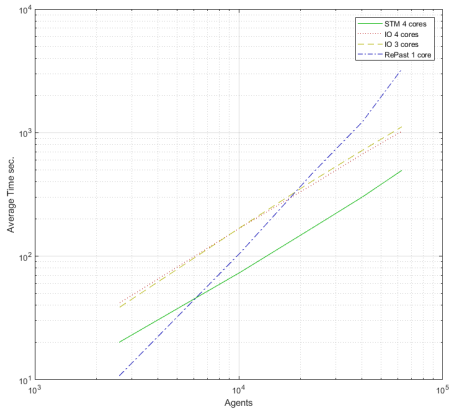
4.3 Varying Grid Size, Constant Cores

In this experiment we varied the grid size and used constantly 4 cores. Because in the previous experiment *Lock-Based* performed best on 3 cores, we additionally ran Lock-Based on 3 cores as well. Note that the RePast experiments all ran on a single (1) core and were conducted to have a rough estimate where the functional approach is in comparison to the imperative. The results are reported in Table 3 and plotted in Figure 4.

It is clear that the *STM* implementation outperforms the *Lock-Based* implementation by a substantial factor. Surprisingly, the *Lock-Based* implementation on 4 core scales just slightly better with increasing agents number than on 3 cores, something we wouldn't have anticipated based on the results seen in Table 2. Also while on a 51x51 grid the single (1) core Java *RePast* version outperforms the 4 core Haskell *STM* version by a factor of 2. The figure is inverted on a 251x251 grid where the 4 core Haskell *STM* version outperforms



(a) Normal Scale



(b) Logarithmic scale on both axes

Fig. 4. Performance on varying grid sizes.

Grid-Size	Commits	Retries	Ratio
51 x 51 (2,601)	2,601,000	1306.5	0.0
101 x 101 (10,201)	10,201,000	3712.5	0.0
151 x 151 (22,801)	22,801,000	8189.5	0.0
201 x 201 (40,401)	40,401,000	13285.0	0.0
251 x 251 (63,001)	63,001,000	21217.0	0.0

Table 4. Retry ratios on varying grid sizes on 4 cores.

the single core Java *Repast* version by a factor of 6. This might not be entirely surprising because we compare single (1) core against multi-core performance - still the scaling is indeed impressive and we would not have anticipated an increase of factor 6.

4.4 Retries

Of very much interest when using STM is the retry-ratio, which obviously depends highly on the read-write patterns of the respective model. We used the *stm-stats* library to record statistics of commits, retries and the ratio. The results are reported in Table 4.

Independent of the number of agents we always have a retry-ratio of 0.0. This indicates that this model is *very* well suited to STM, which is also directly reflected in the much better performance over the *Lock-Based* implementation. Obviously this ratio stems from the fact, that in our implementation we have *very* few writes, which happen only in case when an agent changes from Susceptible to Infected or from Infected to Recovered.

4.5 Going Large-Scale

To test how far we can scale up the number of cores in both the *Lock-Based* and *STM* cases, we ran two experiments, 51x51 and 251x251, on Amazon S2 instances with a larger number

	Cores	51x51	251x251
Lock-Based	16	72.5	1830.5
	32	73.1	1882.2
STM	16	**8.6**	**237.0**
	32	12.0	248.7

Table 5. Performance on varying cores on Amazon S2 Services.

of cores than our local machinery, starting with 16 and 32 to see if we are running into decreasing returns. The results are reported in Table 5.

As expected, the *Lock-Based* approach doesn't scale up to many cores because each additional core brings more contention to the lock, resulting in an even more decreased performance. This is particularly obvious in the 251x251 experiment because of the much larger number of concurrent agents. The *STM* approach returns better performance on 16 cores but fails to scale further up to 32 where the performance drops below the one with 16 cores. In both STM cases we measured a retry-ratio of 0, thus we assume that with 32 cores we become limited by the overhead of STM transactions [30] because the workload of an STM action in our SIR implementation is quite small.

4.6 Discussion

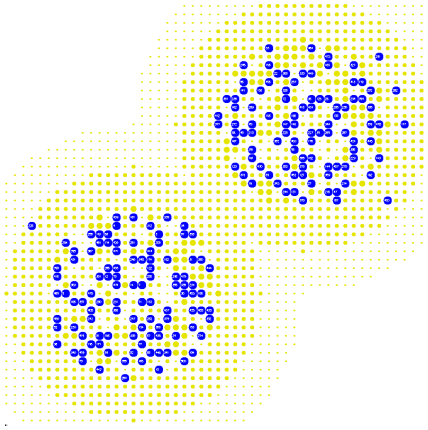
Reflecting of the performance data leads to the following insights:

(1) Running in STM and sharing state using a transactional variable is much more time-efficient than a *Sequential* approach but potentially sacrifices determinism: repeated runs might not lead to same dynamics despite same initial conditions.
(2) Running STM on multiple cores concurrently *does* lead to a significant performance improvement over the *Sequential* implementation for that case-study.
(3) *STM* outperforms the *Lock-Based* implementation substantially and scales much better to multiple cores.
(4) *STM* on single (1) core is still about half as fast as an object-oriented Java *RePast* implementation on a single (1) core.
(5) *STM* on multiple cores dramatically outperforms the single (1) core object-oriented Java *RePast* implementation on a single (1) core on instances with large agent numbers and scales much better to increasing number of agents.

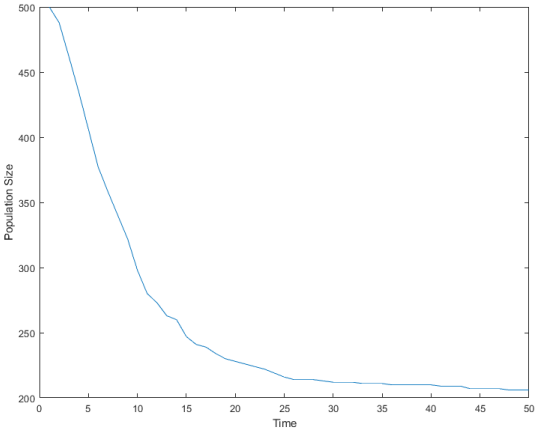
5 CASE STUDY 2: SUGARSCAPE

One of the first models in Agent-Based Simulation was the seminal Sugarscape model developed by Epstein and Axtell in 1996 [7]. Their aim was to *grow* an artificial society by simulation and connect observations in their simulation to phenomenon observed in real-world societies. In this model a population of agents move around in a discrete 2D environment, where sugar grows, and interact with each other and the environment in many different ways. The main features of this model are (amongst others): searching, harvesting and consuming of resources, wealth and age distributions, population dynamics under sexual reproduction, cultural processes and transmission, combat and assimilation, bilateral decentralized trading (bartering) between agents with endogenous demand and supply, disease processes transmission and immunology.

We implemented the *Carrying Capacity* (p. 30) section of Chapter II of the book [7]. There, in each step agents search (move) to the cell with the most sugar they see within



(a) Visualisation of the Sugarscape at $t = 50$



(b) Dynamics population size over 50 steps

Fig. 5. Visualisation of our SugarScape implementation and dynamics of the population size over 50 steps. The white numbers in the blue agent circles are the agents unique ids.

their vision, harvest all of it from the environment and consume sugar because of their metabolism. Sugar regrows in the environment over time. Only one agent can occupy a cell at a time. Agents don't age and cannot die from age. If agents run out of sugar due to their metabolism, they die from starvation and are removed from the simulation. The authors report that the initial number of agents quickly drops and stabilises around a level depending on the model parameters. This is in accordance with our results as we show in Figure 5 and guarantees that we don't run out of agents. The model parameters are as follows:

- Sugar Endowment: each agent has an initial sugar endowment randomly uniform distributed between 5 and 25 units;
- Sugar Metabolism: each agent has a sugar metabolism randomly uniform distributed between 1 and 5;
- Agent Vision: each agent has a vision randomly uniform distributed between 1 and 6, same for each of the 4 directions (N, W, S, E);
- Sugar Growback: sugar grows back by 1.0 unit per step until the maximum capacity of a cell is reached;
- Agent Number: initially 500 agents;
- Environment Size: 50 x 50 cells with toroid boundaries which wrap around in both x and y dimension.

5.1 Experiment Design

We compare four different implementations [3]:

(1) Sequential - All agents are run after another (including the environment) and the environment is shared amongst the agents using a read/write state context.

[3]The code is freely available at https://github.com/thalerjonathan/phd/tree/master/public/stmabs/code/SugarScape

(2) Lock-Based - All agents are run concurrently and the environment is shared using a global reference amongst the agents which acquire and release a lock when accessing it.
(3) STM TVar - All agents are run concurrently and the environment is shared using a *TVar* amongst the agents.
(4) STM TArray - All agents are run concurrently and the environment is shared using a *TArray* amongst the agents.

The model specification requires to shuffle agents before every step (Footnote 12 on page 26 [7]). In the *Sequential* approach we do this explicitly but in the *Lock-Based* and both *STM* approaches this happens automatically due to race-conditions in concurrency thus we arrive at an effectively shuffled processing of agents: we implicitly assume that the order of the agents is *effectively* random in every step. Not making assumptions of the ordering of thread execution is a core principle of concurrent programming, which we exploit in this case to assume an effective randomness. The important difference between the two approaches is that in the *Sequential* approach we have full control over this randomness but in the *STM* not - also this means that repeated runs with the same initial conditions might lead to slightly different results.

Note that in the concurrent implementations we have two options for running the environment: either asynchronously as a concurrent agent at the same time with the population agents or synchronously after all agents have run. We must be careful though as running the environment as a concurrent agent can be seen as conceptually wrong because the time when the regrowth of the sugar happens is now completely random. In this case it could happen that sugar regrows in the very first transaction or in the very last, different in each step, which can be seen as a violation of the model specifications. Thus we do not run the environment concurrently with the agents but synchronously after all agents have run.

We follow [22] and measure the average number of steps per second of the simulation over 60 seconds. For each experiment we conducted 8 runs on our machine (see Table 1) under no additional work-load and report the average. In the experiments we varied the number of cores when running concurrently - the numbers are always indicated clearly.

Note that we omit the graphical rendering in the functional approach because it is a serious bottleneck taking up substantial amount of the simulation time. Although visual output is often important in ABS, it is not what we are interested here thus we completely omit it and only output the number of agents in the simulation at each step piped into a file, thus omitting slow output to the console [4].

5.2 Constant Agent Size

In a first approach we compare the performance of all implementations on varying numbers of cores. The results are reported in Table 6 and plotted in Figure 6.

As expected, the *Sequential* implementation is the slowest, followed by the *Lock-Based* and *TVar* approach whereas *TArray* is the best performing one.

We clearly see that using *TVar* to share the environment is a very inefficient choice: *every* write to a cell leads to a retry independent whether the reading agent reads that changed cell or not, because the data-structure can not distinguish between individual cells. By using a *TArray* we can avoid the situation where a write to a cell in a far distant location of the environment will lead to a retry of an agent which never even touched that cell. Also the

[4]Note that we need to produce *some* output because of Haskells laziness - if we wouldn't output anything from the simulation then the expressions would actually never be fully evaluated thus resulting in high number of steps per second but which obviously don't really reflect the true computations done.

	Cores	Steps	Retries
Sequential	1	39.4	N/A
Lock-Based	1	43.0	N/A
	2	51.8	N/A
	3	57.4	N/A
	4	58.1	N/A
STM *TVar*	1	**47.3**	0.0
	2	53.5	1.1
	3	57.1	2.2
	4	53.0	3.2
STM *TArray*	1	45.4	0.0
	2	**65.3**	0.02
	3	**75.7**	0.04
	4	**84.4**	0.05

Table 6. Steps per second and retries on 50x50 grid with 500 initial agents on varying cores.

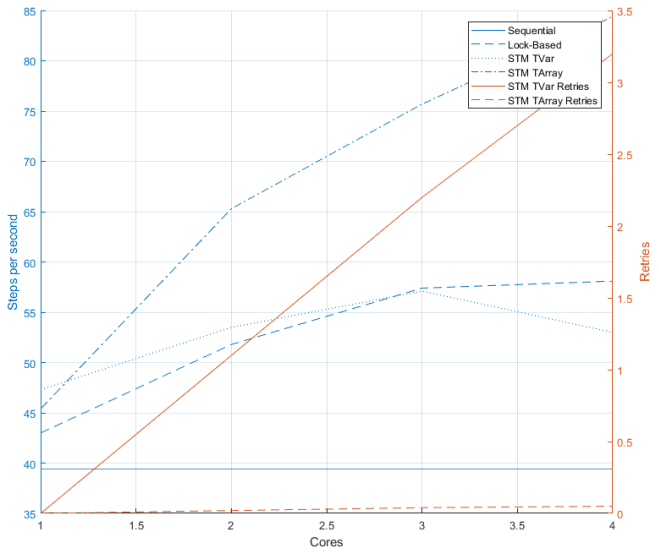


Fig. 6. Steps per second and retries on 50x50 grid and 500 initial agents on varying cores.

TArray seems to scale up by 10 steps per second for every core added. It will be interesting to see how far this could go with the Amazon experiment, as we seem not to hit a limit with 4 cores yet.

The inefficiency of *TVar* is also reflected in the nearly similar performance of the *Lock-Based* implementation which even outperforms it on 4 cores. This is due to very similar approaches because both operate on the whole environment instead of only the cells as *TArray* does. This seems to be a bottleneck in *TVar* reaching the best performance on 3

Agents	Sequential	Lock-Based	TVar (3 cores)	TVar (4 cores)	TArray
500	14.4	20.2	20.1	18.5	**71.9**
1,000	6.8	10.8	10.4	9.5	**54.8**
1,500	4.7	8.1	7.9	7.3	**44.1**
2,000	4.4	7.6	7.4	6.7	**37.0**
2,500	5.3	5.4	9.2	8.9	**33.3**

Table 7. Steps per second on 50x50 grid with varying number of agents with 4 (and 3) cores except Sequential (1 core).

cores, which then drops on 4 cores. The *Lock-Based* approach seems to reduce its returns on increased number of cores hitting a limit at 4 cores as well.

5.3 Scaling up Agents

So far we kept the initial number of agents at 500, which due to the model specification, quickly drops and stabilises around 200 due to the carrying capacity of the environment as described in the book [7] section *Carrying Capacity* (p. 30).

We now want to see performance of our approaches under increased number of agents. For this we slightly change the implementation: always when an agent dies it spawns a new one which is inspired by the ageing and birthing feature of Chapter III in the book [7]. This ensures that we keep the number of agents roughly constant (still fluctuates but doesn't drop to low levels) over the whole duration. This ensures a constant load of concurrent agents interacting with each other and demonstrates also the ability to terminate and fork threads dynamically during the simulation.

Except for the *Sequential* approach we ran all experiments with 4 cores (TVar with 3 as well). We looked into the performance of 500, 1,000, 1,500, 2,000 and 2,500 (maximum possible capacity of the 50x50 environment). The results are reported in Table 7 and plotted in Figure 7.

As expected, the *TArray* implementation outperforms all others substantially. Also as expected, the *TVar* implementation on 3 cores is faster than on 4 cores as well when scaling up to more agents. The *Lock-Based* approach performs about the same as the *TVar* on 3 cores because of the very similar approaches: both access the *whole* environment. Still the *TVar* approach uses one core less to arrive at the same performance, thus strictly speaking outperforming the *Lock-Based* implementation.

What seems to be very surprising is that in the *Sequential* and *TVar* cases the performance with 2,500 agents is *better* than the one with 2,000 agents. The reason for this is that in the case of 2,500 agents, an agent can't move anywhere because all cells are already occupied. In this case the agent won't rank the cells in order of their pay-off (max sugar) to move to but just stays where it is. We hypothesize that due to Haskells laziness the agents actually never look at the content of the cells in this case but only the number which means that the cells themselves are never evaluated which further increases performance. This leads to the better performance in case of *Sequential* and *TVar* because both exploit laziness. In the case of the *Lock-Based* approach we still arrive at a lower performance because the limiting factor are the unconditional locks. In the case of the *TArray* approach we also arrive at a lower performance because it seems that STM perform reads on the neighbouring cells which are not subject to lazy evaluation.

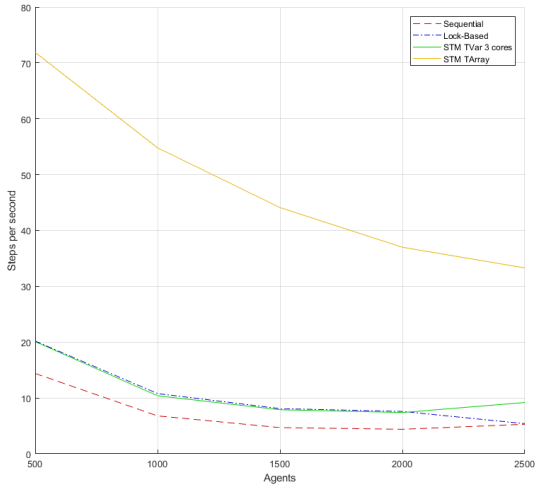


Fig. 7. Steps per second on 50x50 grid and varying number of agents with 4 (and 3) cores except Sequential (1 core).

	Cores	Carrying Capacity	Rebirthing
Lock-Based	16	53.9	4.4
	32	44.2	3.6
STM TArray	16	**116.8** (0.23)	**39.5** (0.08)
	32	**109.8** (0.41)	**31.3** (0.18)

Table 8. Steps per second on varying cores on Amazon S2 Services.

We also measured the average retries both for *TVar* and *TArray* under 2,500 agents where the *TArray* approach shows best scaling performance with 0.01 retries whereas *TVar* averages at 3.28 retries. Again this can be attributed to the better transactional data-structure which reduces retry-ratio substantially to near-zero levels.

5.4 Going Large-Scale

To test how far we can scale up the number of cores in both the *Lock-Based* and *TArray* cases, we ran the two experiments (carrying capacity and rebirthing) on Amazon S2 instances with increasing number of cores starting with 16 and 32 to see if we run into decreasing returns. The results are reported in Table 8.

As expected, the *Lock-Based* approach doesn't scale up to many cores because each additional core brings more contention to the lock, resulting in even more decreased performance. This is particularly obvious in the rebirthing experiment because of the much larger number of concurrent agents. The *TArray* approach returns better performance on 16 cores but fails to scale further up to 32 where the performance drops below the one with 16 cores. We indicated the retry-ratio in brackets and see that they roughly double from 16 to 32, which is the reason why performance drops as at this point.

5.5 Comparison with other approaches

The paper [22] reports a performance of 17 steps in RePast, 18 steps in MASON (both non-parallel) and 2,000 steps per second on a GPU on a 128x128 grid. Although our *Sequential* implementation, which runs non-parallel as well, outperforms the RePast and MASON implementations of [22], one must be very well aware that these results were generated in 2008, on current hardware of that time.

When we run the SugarScape example of RePast with the same model parameters as ours on the same machine (see Table 1) we arrive at roughly 450 steps per second - a factor of about 3.8 faster than even our STM *TArray* implementation on 16 cores. This might seem quite shocking, even more so because RePast also performs visual output, rendering the SugarScape in every step. When scaling up the agents to 2,500 the RePast version arrives around roughly 95 steps per second which is still faster by a factor of 3 than our 4 core *TArray* implementation. We attribute this substantial performance difference to the inherent performance difference of functional programming to imperative approaches as already outlined in the previous section.

The very high performance on the GPU does not concern us here as it follows a very different approach than ours. We focus on speeding up implementations on the CPU as directly as possible without locking overhead. When following a GPU approach one needs to map the model to the GPU which is a delicate and non-trivial matter. With our approach we show that speed up with concurrency is very possible without the low-level locking details or the need to map to GPU. Also some features as bilateral trading between agents, where a pair of agents needs to come to a conclusion over multiple synchronous steps, is difficult to implement on a GPU whereas this is easily possible using STM.

Note that we kept the grid-size constant because we implemented the environment as a single agent which works sequentially on the cells to regrow the sugar. Obviously this doesn't really scale up on parallel hardware and experiments which we haven't included here due to lack of space, show that the performance goes down dramatically when we increase the environment to 128x128 with same number of agents which is the result of Amdahl's law where the environment becomes the limiting factor of the simulation. Depending on the underlying data-structure used for the environment we have two options to solve this problem. In the case of the *Sequential* and *TVar* implementation we build on an indexed array, which we can be updated in parallel using the existing data-parallel support in Haskell. In the case of the *TArray* approach we have no option but to run the update of every cell within its own thread. We leave both for further research as it is out of scope of this paper.

5.6 Discussion

Reflecting on the performance data leads to the following insights:

- Selecting the right transactional data-structure is very model-specific and can lead to dramatically different performance results. In this case the *TArray* performed best due to many writes but in the SIR case-study a *TVar* showed good enough results due to the very low number of writes.
- A *TArray* might come with an overhead, performing worse on low number of cores than a *TVar* approach but has the benefit of quickly scaling up to multiple cores.
- When not carefully selecting the right transactional data-structure which supports fine-grained concurrency, a lock-based implementation might perform as well or even outperform the STM approach as can be seen when using the *TVar*.

- Depending on the transactional data-structure, scaling up to multiple cores hits a limit at some point. In the case of the *TVar* the best performance is reached with 3 cores. With the *TArray* we reached this limit around 16 cores.
- A well implemented STM approach with a carefully selected transactional data-structure consistently outperforms the lock-based approach and scales up to multiple cores considerably better.
- Unfortunately, for this model the performance is nowhere comparable to imperative approaches, which we attribute to the inherent performance difference of functional programming to imperative approaches. With the use of advanced language features we might arrive at much improved performance but we leave this for further research as we focus primarily on the comparison between lock-based and STM approaches.

6 CONCLUSION

In this paper we investigated the potential for using STM for parallel, large-scale ABS and come to the conclusion that it is indeed a very promising alternative over lock-based approaches as our case-studies have shown. The STM approaches all consistently outperformed the lock-based implementations and scaled much better to larger number of CPUs. Besides, the concurrency abstractions of STM are very powerful, yet simple enough to allow convenient implementation of concurrent agents without the problems of lock-based implementation. Due to most ABS being primarily pure computations, which do not need interactive input from the user or files / network during simulation, the fact that no such interactions can occur within an agent when running within STM is not a problem.

Interestingly, STM primitives map nicely to ABS concepts. When having a shared environment, it is natural either using *TVar* or *TArray*, depending on the environments nature. Also, there exists the *TChan* primitive, which can be seen as a persistent message box for agents, underlining the message-oriented approach found in many agent-based models [2, 38]. Also *TChan* offers a broadcast transactional channel, which supports broadcasting to listeners which maps nicely to a pro-active environment or a central auctioneer upon which agents need to synchronize. The benefits of these natural mappings are that using STM takes a big portion of burden from the modeller as one can think in STM primitives instead of low level locks and concurrent operational details.

The strong static type-system of Haskell adds another benefit. By running in the STM instead of IO context makes the concurrent nature more explicit and at the same time restricts it to purely STM behaviour. So despite obviously losing the reproducibility property due to concurrency, we still can guarantee that the agents can't do arbitrary IO as they are restricted to STM operations only.

Depending on the nature of the transactions, retries could become a bottle neck, resulting in a live lock in extreme cases. The central problem of STM is to keep the retries low, which is directly influenced by the read/writes on the STM primitives. By choosing more fine-grained / suitable data-structures e.g. using a *TArray* instead of an indexed array within a *TVar*, one can reduce retries and increase performance significantly and avoid the problem of live locks as we have shown.

After the strong performance results of the SIR case-study in Section 4 we come to the conclusion, that the performance results of the SugarScape case-study are not as compelling. This shows that for some ABS models, performance in a concurrent multi-core functional implementation is still nowhere near the established single-core imperative implementations in e.g. RePast. This does not come as a surprise because as already pointed out in previous sections, functional programming in general is not as fast as imperative approaches and

involves much more experience and sophisticated techniques to arrive at performance of imperative approaches [19].

Despite the indisputable benefits of using STM within a pure functional setting like Haskell, it exists also in other imperative languages (Python, Java and C++, etc) and we hope that our research sparks interest in the use of STM in ABS in general and that other researchers pick up the idea and apply it to the established imperative languages Python, Java, C++ in the ABS community as well.

7 FURTHER RESEARCH

So far we only implemented a tiny bit of the Sugarscape model and left out the later chapters which are more involved as they incorporate direct synchronous communication between agents. Such mechanisms are very difficult to approach in GPU based approaches [22] but should be quite straightforward in STM using *TChan* and retries.

We have not focused on implementing an approach like *Sense-Think-Act* cycle as mentioned in [39]. This could offer lot of potential for parallelisation due to sense and think happening isolated from each agent, without interfering with global shared data. We expect additional speed-up from such an approach.

So far we only looked at time-driven models (note that the Sugarscape is basically a time-driven model where each agent acts in each step) where all agents run concurrently in lock-step. Such models are very well suited to concurrent execution because normally in such models there is no restriction on the order of agents imposed and if then it is almost always to be random, making it perfect for concurrent execution. It would be of interest whether we can apply STM and concurrency to an event-driven approach as well. Generally one could run agents concurrently and undo actions when there are inconsistencies - something which pure functional programming in Haskell together with STM supports out of the box. Such an approach could be theoretically implemented using Parallel Discrete Event Simulation (PDES) [8] and it would be interesting to see how an event-driven ABS approach based on an underlying PDES implementation would perform.

Our implementations focus only on local parallelisation and concurrency and avoided distributed computing as it was out of the scope of this paper. It would be of interest to see how we can map functional ABS to distributed computing using a message-based approach found in the Cloud Haskell framework.

ACKNOWLEDGMENTS

The authors would like to thank J. Hey and M. Handley for constructive feedback, comments and valuable discussions.

REFERENCES

[1] Sameera Abar, Georgios K. Theodoropoulos, Pierre Lemarinier, and Gregory M.P. O'Hare. 2017. Agent Based Modelling and Simulation tools: A review of the state-of-art software. *Computer Science Review* 24 (May 2017), 13–33. https://doi.org/10.1016/j.cosrev.2017.03.001

[2] Gul Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA.

[3] Nikolaos Bezirgiannis. 2013. *Improving Performance of Simulation Software Using Haskells Concurrency & Parallelism.* Ph.D. Dissertation. Utrecht University - Dept. of Information and Computing Sciences.

[4] Franco Cicirelli, Andrea Giordano, and Libero Nigro. 2015. Efficient Environment Management for Distributed Simulation of Large-scale Situated Multi-agent Systems. *Concurr. Comput. : Pract. Exper.* 27, 3 (March 2015), 610–632. https://doi.org/10.1002/cpe.3254

[5] Anthony Discolo, Tim Harris, Simon Marlow, Simon Peyton Jones, and Satnam Singh. 2006. Lock Free Data Structures Using STM in Haskell. In *Proceedings of the 8th International Conference on*

Functional and Logic Programming (FLOPS'06). Springer-Verlag, Berlin, Heidelberg, 65–80. https://doi.org/10.1007/11737414_6

[6] Richard H. Enns. 2010. *It's a Nonlinear World* (1st ed.). Springer Publishing Company, Incorporated.

[7] Joshua M. Epstein and Robert Axtell. 1996. *Growing Artificial Societies: Social Science from the Bottom Up*. The Brookings Institution, Washington, DC, USA.

[8] Richard M. Fujimoto. 1990. Parallel Discrete Event Simulation. *Commun. ACM* 33, 10 (Oct. 1990), 30–53. https://doi.org/10.1145/84537.84545

[9] Les Gasser and Kelvin Kakugawa. 2002. MACE3J: Fast Flexible Distributed Simulation of Large, Large-grain Multi-agent Systems. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems: Part 2 (AAMAS '02)*. ACM, New York, NY, USA, 745–752. https://doi.org/10.1145/544862.544918 event-place: Bologna, Italy.

[10] B. Kaan Gorur, Kayhan Imre, Halit Oguztuzun, and Levent Yilmaz. 2016. Repast HPC with Optimistic Time Management. In *Proceedings of the 24th High Performance Computing Symposium (HPC '16)*. Society for Computer Simulation International, San Diego, CA, USA, 4:1–4:9. https://doi.org/10.22360/SpringSim.2016.HPC.046 event-place: Pasadena, California.

[11] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. 2005. Composable Memory Transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '05)*. ACM, New York, NY, USA, 48–60. https://doi.org/10.1145/1065944.1065952

[12] Tim Harris and Simon Peyton Jones. 2006. Transactional memory with data invariants. https://www.microsoft.com/en-us/research/publication/transactional-memory-data-invariants/

[13] Joshua Hay and Philip A. Wilsey. 2015. Experiments with Hardware-based Transactional Memory in Parallel Simulation. In *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM PADS '15)*. ACM, New York, NY, USA, 75–86. https://doi.org/10.1145/2769458.2769462 event-place: London, United Kingdom.

[14] Armin Heindl and Gilles Pokam. 2009. Modeling Software Transactional Memory with AnyLogic. In *Proceedings of the 2Nd International Conference on Simulation Tools and Techniques (Simutools '09)*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium, 10:1–10:10. https://doi.org/10.4108/ICST.SIMUTOOLS2009.5581

[15] Jan Himmelspach and Adelinde M. Uhrmacher. 2007. Plug'n Simulate. In *40th Annual Simulation Symposium (ANSS'07)*. 137–143. https://doi.org/10.1109/ANSS.2007.34 ISSN: 1080-241X.

[16] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. 2003. Arrows, Robots, and Functional Reactive Programming. In *Advanced Functional Programming*, Johan Jeuring and Simon L. Peyton Jones (Eds.). Number 2638 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, 159–187. https://doi.org/10.1007/978-3-540-44833-4_6

[17] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. 2007. A History of Haskell: Being Lazy with Class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages (HOPL III)*. ACM, New York, NY, USA, 12–1–12–55. https://doi.org/10.1145/1238844.1238856

[18] W. O. Kermack and A. G. McKendrick. 1927. A Contribution to the Mathematical Theory of Epidemics. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 115, 772 (Aug. 1927), 700–721. https://doi.org/10.1098/rspa.1927.0118

[19] Chris kqr. 2017. On Competing with C Using Haskell. https://two-wrongs.com/on-competing-with-c-using-haskell

[20] Michael Lees, Brian Logan, and Georgios Theodoropoulos. 2008. Using Access Patterns to Analyze the Performance of Optimistic Synchronization Algorithms in Simulations of MAS. *Simulation* 84, 10-11 (Oct. 2008), 481–492. https://doi.org/10.1177/0037549708096691

[21] B. Logan and G. Theodoropoulos. 2001. The distributed simulation of multiagent systems. *Proc. IEEE* 89, 2 (Feb. 2001), 174–185. https://doi.org/10.1109/5.910853

[22] Mikola Lysenko and Roshan M. D'Souza. 2008. A Framework for Megascale Agent Based Model Simulations on Graphics Processing Units. *Journal of Artificial Societies and Social Simulation* 11, 4 (2008), 10. http://jasss.soc.surrey.ac.uk/11/4/10.html

[23] Charles M. Macal. 2010. To Agent-based Simulation from System Dynamics. In *Proceedings of the Winter Simulation Conference (WSC '10)*. Winter Simulation Conference, Baltimore, Maryland, 371–382. http://dl.acm.org/citation.cfm?id=2433508.2433551

[24] C. M. Macal. 2016. Everything you need to know about agent-based modelling and simulation. *Journal of Simulation* 10, 2 (May 2016), 144–156. https://doi.org/10.1057/jos.2016.7

[25] Simon Marlow, Simon Peyton Jones, and Satnam Singh. 2009. Runtime Support for Multicore Haskell. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09)*. ACM, New York, NY, USA, 65–78. https://doi.org/10.1145/1596550.1596563

[26] Ruth Meyer. 2014. Event-Driven Multi-agent Simulation. In *Multi-Agent-Based Simulation XV (Lecture Notes in Computer Science)*. Springer, Cham, 3–16. https://doi.org/10.1007/978-3-319-14627-0_1

[27] R. Minson and G. K. Theodoropoulos. 2008. Distributing RePast agent-based simulations with HLA. *Concurrency and Computation: Practice and Experience* 20, 10 (2008), 1225–1256. https://doi.org/10.1002/cpe.1280

[28] Henrik Nilsson, Antony Courtney, and John Peterson. 2002. Functional Reactive Programming, Continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell (Haskell '02)*. ACM, New York, NY, USA, 51–64. https://doi.org/10.1145/581690.581695

[29] Michael J. North, Nicholson T. Collier, Jonathan Ozik, Eric R. Tatara, Charles M. Macal, Mark Bragen, and Pam Sydelko. 2013. Complex adaptive systems modeling with Repast Simphony. *Complex Adaptive Systems Modeling* 1, 1 (March 2013), 3. https://doi.org/10.1186/2194-3206-1-3

[30] Cristian Perfumo, Nehir Sönmez, Srdjan Stipic, Osman Unsal, Adrián Cristal, Tim Harris, and Mateo Valero. 2008. The Limits of Software Transactional Memory (STM): Dissecting Haskell STM Applications on a Many-core Environment. In *Proceedings of the 5th Conference on Computing Frontiers (CF '08)*. ACM, New York, NY, USA, 67–78. https://doi.org/10.1145/1366230.1366241

[31] Patrick F. Riley and George F. Riley. 2003. Next Generation Modeling III - Agents: Spades — a Distributed Agent Simulation Environment with Software-in-the-loop Execution. In *Proceedings of the 35th Conference on Winter Simulation: Driving Innovation (WSC '03)*. Winter Simulation Conference, 817–825. http://dl.acm.org/citation.cfm?id=1030818.1030926 event-place: New Orleans, Louisiana.

[32] Nir Shavit and Dan Touitou. 1995. Software Transactional Memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '95)*. ACM, New York, NY, USA, 204–213. https://doi.org/10.1145/224964.224987

[33] Vinoth Suryanarayanan and Georgios Theodoropoulos. 2013. Synchronised Range Queries in Distributed Simulations of Multiagent Systems. *ACM Trans. Model. Comput. Simul.* 23, 4 (Nov. 2013), 25:1–25:25. https://doi.org/10.1145/2517449

[34] Vinoth Suryanarayanan, Georgios Theodoropoulos, and Michael Lees. 2013. PDES-MAS : distributed simulation of multi-agent systems. (2013). https://doi.org/10.1016/j.procs.2013.05.231

[35] Jonathan Thaler, Thorsten Altenkirch, and Peer-Olaf Siebers. 2018. Pure Functional Epidemics: An Agent-Based Approach. In *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages (IFL 2018)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/3310232.3310372 event-place: Lowell, MA, USA.

[36] Jonathan Thaler and Peer-Olaf Siebers. 2017. The Art Of Iterating: Update-Strategies in Agent-Based Simulation. Dublin.

[37] Uri Wilensky and William Rand. 2015. *An Introduction to Agent-Based Modeling: Modeling Natural, Social, and Engineered Complex Systems with NETLogo*. MIT Press. https://www.amazon.co.uk/Introduction-Agent-Based-Modeling-Natural-Engineered/dp/0262731894

[38] Michael Wooldridge. 2009. *An Introduction to MultiAgent Systems* (2nd ed.). Wiley Publishing.

[39] Jiajian Xiao, Philipp Andelfinger, David Eckhoff, Wentong Cai, and Alois Knoll. 2018. A Survey on Agent-based Simulation using Hardware Accelerators. *arXiv:1807.01014 [cs]* (July 2018). http://arxiv.org/abs/1807.01014 arXiv: 1807.01014.

Received October 2018