RESEARCH

A tale of lock-free Agents: Towards Software Transactional Memory in parallel Agent-Based Simulation

Jonathan Thaler* and Peer-Olaf Siebers

Correspondence:
onathan.thaler@nottingham.ac.uk
University of Nottingham, 7301
Wollaton Rd, NG8 1BB
Nottingham, UK
Full list of author information is
available at the end of the article

Abstract

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 acknowledged 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 an excess 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, in which 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 CPU cores 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 CPU cores both on local hardware and on Amazon EC. Further, by utilizing the pure functional language Haskell we gain the benefits of immutable data and lack of unrestricted side effects guaranteed at compile-time, making validation easier and leading to increased confidence in the correctness of an implementation, something of fundamental importance and benefit in parallel programming in general and scientific computing like ABS in particular.

Keywords: Agent-Based Simulation; Software Transactional Memory; Parallel Programming; Haskell

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 to go massively parallel due to availability of cheap 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* [1] was called out and as a substantial body of research for parallel ABS shows [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13].

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 [4]. Thus, the problem of parallelising ABS boils down to the problem of how to synchronise access to shared state without violating the causality principle and resource constraints [3, 2]. Various researchers have developed different techniques, where most of them are 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 ABS is challenging as agents are autonomous, which means that the topology can change in every step, making it hard to predict the topology of logical processes in advance [4], posing a difficult problem for parallelisation in general [13]. The work [2, 5] 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 [3] 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 ABS have been developed and their internals published in research papers: the SPADES system [6] manages agents through UNIX pipes using a parallel sense-think-act cycle employing a conservative PDES approach; Mace3J [7] a Java based system running on single- or multicore workstations implements a message passing approach to parallelism; James II [8] is also a Java based system and focuses on PDEVS simulation with a plugin architecture to facilitate reuse of models; the well known RePast-HPC [9, 10] framework is 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 [12]. Further, this sophisticated and powerful machinery is not always required as ABS 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 ABS frameworks could be used for this, they are an excess and more straightforward concepts for parallelising ABS would be appropriate. However, for this case there does not exist much research, and implementers either resort to the distributed ABS frameworks, implement their own low-level concurrency plumbing, which can be considerably complex - or simply refrain 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 the 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++, don't make adding parallelism easy, due to their inherent use of unrestricted side effects. Further, they mostly follow a lock-based approach to concurrency which is error prone and does not compose.

This paper proposes Software Transactional Memory (STM) in conjunction with the functional programming language Haskell [14] as a new underlying concept for local parallelisation of ABS. We hypothesise that by using STM in Haskell, implementing local parallel ABS is considerably easier than with lock-based approaches, less error prone and easier to validate. Although STM exists in other languages as well by now, 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. Our hypothesis is supported by [15], which gives a good indication of 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.

To the best of our knowledge we are the first to *systematically* discuss the use of STM in the context of ABS. However, the idea of applying transactional memory to simulation in general is not new and was already explored in the work [11], 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.

The master thesis [16] investigates Haskell's parallel and concurrency features to implement (amongst others) *HLogo*, a Haskell clone of the NetLogo [17] 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 side effect context (known as `IO`, see more below in section 2.2.1) and therefore can also make use of STM functionality. The benchmarks show that this approach does indeed result in a speedup especially under larger agent populations. Despite the parallelism aspect our work share, our approach is rather different: we avoid unrestricted side effects through `IO` within the agents and explore the use of STM more generally rather than implementing an ABS library.

The aim of this paper is to experimentally investigate the benefits of using STM over lock-based approaches for concurrent ABS models. Therefore, we follow [15] and compare the performance of lock-based and STM implementations and expect that the reduced complexity and increased performance will be directly applicable to ABS as well. We present two case studies in which we employ an agent-based spatial SIR [18, 19] and the well known SugarScape [20] 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 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 system or the error-prone low-level locking semantics of a custom built parallel implementation. Therefore, our paper directly addresses the *Large-scale ABMS* challenge [1], which focuses on efficient modelling and simulating large-scale ABS. Further, using STM, which restricts side effects, and makes parallelism easier, can help in the validation challenge [1] *H5: Requirement that all models be completely validated*.

We start with Section 2, where we discuss the concepts of STM and side effects in 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 model, 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 [21] 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, as an example it is impossible to compose two functions 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 - this means that 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 if it had a consistent view to the shared data by verifying whether other threads have written to memory it has read or not.

However, STM does not come without issues. The authors of [22] analyse several Haskell STM programs with respect to their transactional behaviour and 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 Software Transactional Memory in Haskell

In our case studies we are using the functional programming language Haskell. The paper of [14] 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. With referential transparency a computation does not depend on its context within the system but will produce the same result when run repeatedly with similar inputs, which makes understanding and debugging much easier. Examples for side effects are (amongst others): modifying a global variable, awaiting an input from the keyboard, reading or writing to a file, opening 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, for example 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 which tries to read from a file in a function which only allows drawing random numbers will fail to compile.

In this paper 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 input/output (IO) 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 more in detail below in sections 2.2.3 and 2.2.4.

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 function with side effects 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.

A function without any side effect is called *pure*. Calling a *pure* function 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 by 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 can be run 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 on the other hand is potentially nondeterministic because of nondeterministic 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, 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 [23].

2.2.3 Software Transactional Memory

The work of [24, 25] 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. 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 we lose static guarantees 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, prevents uncontrolled effects which cannot be rolled back.

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 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 or full. To read from an empty or write to a full `TMVar` will cause the current thread to block and retry its transaction when the `TMVar` was updated by another thread.

To execute an `STM` action the function `atomically :: STM a → IO a` is provided, which performs a series of `STM` actions atomically within an `IO` context. It takes the `STM` action which returns a polymorphic value of type `a` and returns an `IO` action which returns a value of type `a`.

2.2.4 STM examples

We provide two examples to demonstrate the use and semantics of STM. The first example is an implementation of the aforementioned functionality, where money is withdrawn from one account and transferred to another. The implementing function `transferFunds` takes two `TVar`, holding the account balances, and the amount to exchange. It executes using `atomically`, therefore running in the `IO` context. It uses the two functions `withdraw` and `deposit` which do the work of withdrawing some amount from one account and depositing some amount to another. This example demonstrates how easy STM can be used: the implementation looks quite straightforward, simply swapping values, without any locking involved or special handling of concurrency, other than the use of `atomically`.

```
transferFunds :: TVar Integer -> TVar Integer -> Integer -> IO ()
transferFunds from to n = atomically $ do
  withdraw from n
  deposit to n

withdraw :: TVar Integer -> Integer -> STM ()
withdraw account amount = do
  balance <- readTVar account
  writeTVar (balance - amount)

deposit :: TVar Integer -> Integer -> STM ()
deposit account amount = do
  balance <- readTVar account
  writeTVar (balance + amount)
```

In the second example we show the retry semantics of STM, by combining the `STM` context with a `StateT` context. A `StateT` context allows to read and write some state, available to the function, which in this example we simply set to be an `Int` value. The combination of both contexts is reflected in the type of the function, which bedsides taking a transactional variable `TVar` holding an `Int`, is `StateT Int STM Int` which means that the function has access to both the `StateT` and `STM` functionality. The first `Int` indicates that the `StateT` context allows to read and write an `Int` value, available to the function; the second `Int` indicates that the function is also an `STM` action and will return an `Int` value.

```
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, therefore retry: block this thread
    -- until the TVar v is written by another thread, then
    -- try again
    then lift retry
    -- condition met: return content ot TVar
    else return n
```

When `stmAction` is run, it prints an `'increment!'` debug message to the console and increments the value in the `StateT` context. 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`. To run `stmAction` we need to spawn a thread:

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

The thread first runs the `StateT` context using the effect runner function `runStateT` which takes the `stmAction` and the initial value of the effect context. This results in an `STM` computation, which is executed through `atomically`. Finally, the result is printed 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 update 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 every increment
    threadDelay 10000
    -- write new value to TVar using atomically, will cause the STM
    -- thread to wake up and retry
    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 always rolled back when a retry is happening. The rollback is easily possible in pure functional programming due to persistent data structures, by simply throwing away the new value and retrying with the old 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 by the Haskell implementation, to support debugging of pure functions. If it would not have been masked as pure, the compiler would have not accepted the program, because the `STM` context does not allow the execution of `IO` actions.

3 Software Transactional Memory in Agent-Based Simulation

In this section we give a short overview of how to apply STM in 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 [26] 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, for example 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

A detailed discussion of how to add STM to agents on a technical level is beyond the focus of this paper as it would require to give an in-depth technical explanation of how our agents are actually implemented [19].

However, the concepts are similar to the example in Section 2.2.4. The agent behaviour is an STM action and has access to the environment either through a TVar or TArray and performs read and write operations directly on it. Each agent itself is run within its own thread, and synchronises with the main thread. Thus, it takes Haskells MVar synchronisation primitives to synchronise with the main thread and simply runs the STM agent behaviour each time it receives the next tick DTime:

```
agentThread :: RandomGen g
            => Int            -- Number of steps to compute
            -> SIRAgent g     -- Agent behaviour
            -> g              -- Random-number generator of the agent
            -> MVar SIRState  -- Synchronisation back to main thread
            -> MVar DTime     -- Receiving DTime for next tick
            -> IO ()
agentThread 0 _ _ _ _ = return () -- all steps computed, terminate thread
agentThread n agent rng retVar dtVar = do
  -- wait for dt to compute current step
  dt <- takeMVar dtVar
  -- compute output of current step
  let agentSTMAction = runAgent agent
  -- run the agents STM action atomically within IO
  ((ret, agent'), rng') <- atomically agentSTMAction
  -- post result to main thread
  putMVar retVar ret
  -- tail recursion to next step
  agentThread (n - 1) agent' rng' retVar dtVar
```

Computing a simulation step is quite 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 compute next tick 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 [27], which allows to simulate the dynamics of an infectious disease like influenza, tuberculosis, chicken pox, rubella and measles spreading through a population [28].

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 [18] but extended it by placing the agents on a discrete 2D grid using a Moore (8 surrounding cells) neighbourhood [19]. A visualisation can be seen in Figure 2.

Due to the continuous-time nature of the SIR model, our implementation follows the time driven [29] 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.

4.1 Experiment Design

In this case study we compare the performance of five (5) implementations under varying numbers of CPU cores and agent numbers. The code of all implementations can be accessed freely from the code repository [30].

1. Sequential - This is the reference implementation as discussed in [19], where the agents are executed sequentially within the main thread without any concurrency. The discrete 2D grid is represented using an indexed array [31] and shared amongst all agents as read-only data, with the main thread updating the array for the next time step.
2. Lock-Based Naive - This is the same implementation as *Sequential*, but the agents now run concurrently in the `IO` context. The discrete 2D grid is also

represented using an indexed array but now modified by the agents themselves and therefore shared using a global reference. The agents acquire and release a lock when accessing the shared environment.

3. Lock-Based Read-Write Lock - This is the same implementation as *Lock-Based Naive*, but uses a read-write lock from concurrent-extra library [32] for a more fine-grained locking strategy. This implementation exploits the fact that in the SIR model, reads outnumber writes by far, making a read-write lock much more appropriate than a naive locking mechanism, which unconditionally acquires and releases the lock. However, it is important to note that this approach works only because the semantics of the model support it: agents read any cells but only write their own cell.
4. Atomic IO - This is the same implementation as *Lock-Based Read-Write Lock* but uses an atomic modification operation to both read and write the shared environment. Although it runs in the `IO` context, it is not a lock-based approach as it does not acquire locks but uses a compare-and-swap hardware instruction. A limitation of this approach is that it is only applicable when there is just a single reference in the program and that all operations need to go through the atomic modification operation. As in the case of the *Lock-Based Read-Write Lock* implementation, this approach works only because the semantics of the model support it.
5. STM - This is the same implementation as *Lock-Based Naive* but agents run in the `STM` context. The discrete 2D grid is also represented using an indexed array but shared amongst all agents through a transactional variable `TVar`.

Each experiment was run on our hardware (see Table 1) under no additional workload until $t = 100$ and stepped using $\Delta t = 0.1$. In the experiments we varied the number of agents (grid size) as well as the number of cores when running concurrently. We checked the visual outputs and the dynamics and they look qualitatively the same as the reference *Sequential* implementation [19]. A rigorous, statistical comparison of all implementations, to investigate the effects of concurrency on the dynamics, is quite involved and therefore beyond the focus of this paper but as a remedy we refer to the use of property-based testing, as shown in [33].

For robust performance measurements we used the microbenchmarking library Criterion [34, 35]. It allows the definition and running of benchmark suites, mea-

Model	Dell XPS 13 (9370)
OS	Ubuntu 19.10 64-bit
RAM	16 GByte
CPU	Intel Core i7-8550U @ 3.6GHz x 8
HD	512Gbyte SSD
Haskell	GHC 8.4.3 (stack resolver lts-12.4)

Table 1: Hardware and software details for all experiments

Cores	Sequential	Lock-Based Naive	Lock-Based Read-Write	Atomic IO	STM
1	73.9 (2.06)	59.2 (0.16)	55.0 (0.22)	**51.0** (0.11)	52.2 (0.23)
2	-	46.5 (0.05)	40.8 (0.18)	**32.4** (0.09)	33.2 (0.03)
3	-	44.2 (0.08)	35.8 (0.06)	**25.5** (0.09)	26.4 (0.05)
4	-	47.4 (0.12)	34.0 (0.32)	**22.7** (0.08)	23.3 (0.19)
5	-	48.1 (0.13)	34.5 (0.06)	**22.6** (0.03)	23.0 (0.06)
6	-	49.1 (0.09)	34.8 (0.03)	**22.3** (0.09)	23.1 (0.05)
7	-	49.8 (0.09)	35.9 (0.15)	**22.8** (0.07)	23.4 (0.22)
8	-	57.2 (0.06	40.4 (0.21)	**25.8** (0.02)	26.2 (0.22)

Table 2: Performance comparison of *Sequential*, *Lock-Based*, *Atomic IO* and *STM* SIR implementations under varying cores with grid size of 51x51 (2,601) agents. Timings in seconds (lower is better), standard deviation in parentheses.

suring performance by executing them repeatedly, fitting actual against expected runtime, reporting mean and standard deviation for statistically robust results. By running each benchmark repeatedly, fitting it using linear regression analysis, Criterion is able to robustly determine whether the measurements fall within a normal range or are outliers (and therefore should be re-run) due to some external influences like additional workload on the machine. Therefore, we made sure to only include measurements Criterion labelled as normal,which meant we re-ran measurements where goodness-of-fit was $R^2 < 0.99$. Criterion ran each of our benchmark 10 times with increasing increments of 1, 2, 3 and 4 times. In the results we report the estimates of ordinary least squares (OLS) regression together with the standard deviation because it gives the most reliable results in terms of statistical robustness.

4.2 Constant Grid Size, Varying Cores

In this experiment we held the grid size constant at 51 x 51 (2,601 agents) and varied the cores where possible. The results are reported in Table 2 and visualised in Figure 3.

Grid Size	Lock-Based Read-Write	Atomic IO	STM
101 x 101 (10,201)	139.0 (0.15)	**91.1** (0.14)	96.5 (0.27)
151 x 151 (22,801)	314.0 (0.67)	**204.0** (0.36)	212.0 (0.16)
201 x 201 (40,401)	559.0 (1.22)	**360.0** (0.61)	382.0 (0.85)
251 x 251 (63,001)	861.0 (0.62)	**571.0** (0.71)	608.0 (1.20)

Table 3: Performance comparison of *Lock-Based Read-Write*, *Atomic IO* and *STM* SIR implementations with varying grid sizes on 4 cores. Timings in seconds (lower is better), standard deviation in parentheses.

Comparing the performance and scaling to multiple cores of the *STM* and both *Lock-Based* implementations shows that the *STM* implementation significantly outperforms the *Lock-Based* ones and scales better to multiple cores. The *Lock-Based* implementations perform best with 3 and 4 cores respective, and shows decreasing performance beyond 4 cores as can be seen in Figure 3. This is no surprise because the more cores, the more contention for the central lock, thus the more likely synchronisation happening, ultimately resulting in reduced performance. This is not an issue in *STM* because no locks are taken in advance due to optimistic locking, where a log of changes is kept allowing the runtime to trigger a retry if conflicting changes are detected upon transacting.

A big surprise however is that the *Atomic IO* implementation is slightly outperforming the *STM* one, which is something we would not have anticipated. We attribute this to the lower overhead of the atomic modification operation.

Both the *STM* and *Atomic IO* implementations are running into decreasing returns after 5 to 6 cores, which we attribute to our hardware. Although virtually it comes across as 8 cores it has only 4 physical ones, implementing hyper threading to simulate 4 additional cores. Due to the fact that resources are shared between two threads of a core, it is only logical that we are running into decreasing returns in all implementations on more than 5 to 6 cores on our hardware.

4.3 Varying Grid Size, Constant Cores

In this experiment we varied the grid size and used constantly 4 cores. 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. However, the *Atomic IO* implementation outperforms

Grid Size	Commits	Retries	Ratio
51 × 51 (2,601)	2,601,000	1306	0.0
101 × 101 (10,201)	10,201,000	3712	0.0
151 × 151 (22,801)	22,801,000	8189	0.0
201 × 201 (40,401)	40,401,000	13285	0.0
251 × 251 (63,001)	63,001,000	21217	0.0

Table 4: Retry ratios of the SIR *STM* implementation with varying grid sizes on 4 cores.

the *STM* one again, where this time the difference is a bit more pronounced due to the higher workload of the experiments.

4.4 Retries

Of very much interest when using STM is the retry ratio, which indicates how many of the total `STM` actions had to be re-run. A high retry ratio shows that a lot of work is wasted on re-running `STM` actions due to many concurrent read and writes. Obviously, it is highly dependent on the read-write patterns of the implementation and indicates how well an STM approach is suitable for the problem at hand. We used the stm-stats [36] 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. 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* implementations. Obviously this ratio stems from the fact, that in our implementation we have *very* few conditional 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 the best performing cases, *Atomic IO* and *STM*, we ran two experiments, 51x51 and 251x251, on an Amazon EC `m5ad.16xlarge` instance with 16 and 32 cores to see if we are running into decreasing returns. The results are reported in Table 5.

The *Atomic IO* implementation is able to scale up performance from 16 to 32 cores in the case of 51x51 but fails to do so with 251x251. We attribute this behaviour to an increased number of retries of the atomic modification operation,

	Cores	51x51	251x251
Atomic IO	16	18.0 (0.21)	638.0 (8.24)
	32	15.6 (0.07)	720.0 (1.70)
STM	16	**14.5** (0.03)	**307.0** (1.12)
	32	**14.7** (0.17)	**269.0** (1.05)

Table 5: Performance comparison of *Atomic IO* and *STM* SIR implementations on 16 and 32 cores on an Amazon EC2 `m5ad.16xlarge` instance. Timings in seconds (lower is better), standard deviations in parentheses.

which obviously increases when the number of agents increases. The *STM* implementation performance on the other hand nearly stays constant on 16 and 32 cores in the 51x51 case. In both cases we measured a retry ratio of 0, thus we conclude that with 32 cores we become limited by the overhead of STM transactions [22] because the workload of an STM action in our SIR implementation is quite small. On the other hand, with heavy load as in the 251x251 case, we see an increased performance with 32 cores.

What is interesting is that on more cores, the *STM* implementations has an edge over the *Atomic IO* approach, and performs better in all cases. It seems that for our problem at hand, the atomic modification operation seems to be not as efficient on many cores as an STM approach.

4.6 Summary

The timing measurements speak a clear language. Running in `STM` and sharing state using a transactional variable `TVar` is much more time efficient than the *Sequential* and both *Lock-Based* approaches. On 5 cores *STM* achieves a speedup factor of 3.2 over the *Sequential* implementation, which is a big improvement compared to the simplicity of the approach. What came as a surprise was that the *Atomic IO* approach slightly outperforms the *STM* implementation. However, the *Atomic IO* approach, which uses an atomic modification operation, is only applicable in case there is just a single reference in the program and requires that all operations go through this atomic modification operation. Whether the latter condition is possible or not, is highly dependent on the model semantics, which support it in the case of the SIR model but unfortunately not in the case of Sugarscape.

Obviously both *Lock-Based*, *Atomic IO* and *STM* sacrifice determinism, which means that repeated runs might not lead to same dynamics despite same initial conditions. However, when sticking to STM, we get the guarantee that the source of this nondeterminism is concurrency within the `STM` context but *nothing else*. This can not be guaranteed in the case of both *Lock-Based* and *Atomic IO* approaches as we lose certain static guarantees when running within the `IO` context. The fact to have *both* a substantial speedup *and* the stronger static guarantees, makes the *STM* approach *very* compelling.

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 [20]. 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 [20]. There, in each step agents search (move) to the cell with the most sugar they see within their vision, harvest all of it from the environment and consume sugar based on 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 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

In this case study we compare the performance of four (4) implementations under varying numbers of CPU cores and agent numbers. The code of all implementations can be accessed freely from the code repository [37].

1 Sequential - This is the reference implementation, where all agents are run after another (including the environment). The environment is represented using an indexed array [31] and shared amongst the agents using a read and write state context.

2 Lock-Based - This is the same implementation as *Sequential*, but all agents are run concurrently within the `IO` context. The environment is also represented as an indexed array but shared using a global reference between the agents which acquire and release a lock when accessing it. Note that the semantics of Sugarscape do not support the implementation of either a read-write lock or atomic modification approach as in the SIR model. In the SIR model, the agents write conditionally to *their own* cell, but this is not the case in Sugarscape, where agents need a consistent view of the whole environment for the whole duration of an agent execution due to the fact that agents do not only write their own locations but also to other locations. If this is not handled correctly, data races are happening and threads overwrite data from other threads, ultimately resulting in incorrect dynamics.

3 STM TVar - This is the same implementation as *Sequential*, but all agents are run concurrently within the `STM` context. The environment is also represented as an indexed array but shared using a `TVar` between the agents.

4 STM TArray - This is the same implementation as *Sequential*, but all agents are run concurrently within the `STM` context. The environment is represented and shared between the agents using a `TArray`.

Ordering The model specification requires to shuffle agents before every step ([20], footnote 12 on page 26). In the *Sequential* approach we do this explicitly but in the *Lock-Based* and both *STM* approaches we assume this to happen automatically due to race conditions from concurrency, thus we arrive at an effectively shuffled processing of agents because we implicitly assume that the order of the agents is *effectively* random in every step. The important difference between the two approaches is that in the *Sequential* approach we have full control over this randomness but in the *STM* and *Lock-Based* not. This has the consequence that repeated runs with the same initial conditions might lead to slightly different results. This decision leaves the execution order of the agents ultimately to Haskell's Runtime System and the underlying operating system. We are aware that by doing this, we make assumptions that the threads run uniformly distributed (fair) but such assumptions should not be made in concurrent programming. As a result we can expect this fact to produces non-uniform distributions of agent runs but we assumed that for this model this does not have a significance influence. In case of doubt, we could resort to shuffling the agents before running them in every step. This problem, where also the influence of nondeterministic ordering on the correctness and results of ABS has to be analysed, deserves in-depth research on its own and is therefore beyond the focus of this paper. As a potential direction for such an investigation, we refer to the technique of property-based testing as shown in [33].

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.

Cores	Sequential	Lock-Based	TVar	TArray
1	25.2 (0.36)	**21.0** (0.12)	21.1 (0.25)	42.0 (2.20)
2	-	**20.0** (0.12)	22.2 (0.21)	24.5 (1.07)
3	-	21.9 (0.19)	23.6 (0.12)	**19.7** (1.05)
4	-	24.0 (0.17)	25.2 (0.16)	**18.9** (0.58)
5	-	26.7 (0.17)	31.0 (0.24)	**20.3** (0.87)
6	-	29.3 (0.57)	35.2 (0.12)	**21.2** (1.49)
7	-	30.0 (0.12)	38.7 (0.42)	**21.0** (0.41)
8	-	31.2 (0.29)	49.0 (0.41)	**21.1** (0.64)

Table 6: Performance comparison of *Sequential*, *Lock-Based*, *TVar* and *TArray* Sugarscape implementations under varying cores with 50x50 environment and 500 initial agents. Timings in seconds (lower is better), standard deviation in parentheses.

The experiment setup is the same as in the SIR case study, with the same hardware (see Table 1), with measurements done under no additional workload using the microbenchmarking library Criterion [34, 35] as well. However, as the Sugarscape model is stepped using natural numbers we ran each measurement until $t = 1000$ and stepped it using $\Delta t = 1$. In the experiments we varied the number of agents as well as the number of cores when running concurrently. We checked the visual outputs and the dynamics and they look qualitatively the same as the reference *Sequential*. As in the SIR case study, a rigorous, statistical comparison of all implementations, to investigate the effects of concurrency on the dynamics, is quite involved and therefore beyond the focus of this paper but as a remedy we refer to the use of property-based testing, as shown in [33].

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, with *TArray* being the fastest one except on 1 and 2 cores, where unexpectedly the *Lock-Based* implementation performed best. Interestingly the *TVar* implementation was the worst performing one of the concurrent implementations.

The reason for the bad performance of *TVar* is that using a `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

Cores	TVar	TArray
1	0.00	0.00
2	1.04	0.02
3	2.15	0.04
4	3.20	0.06
5	4.06	0.07
6	5.02	0.09
7	6.09	0.10
8	8.45	0.11

Table 7: Retry ratio comparison (lower is better) of the *TVar* and *TArray* Sugarscape implementations under varying cores with 50x50 environment and 500 initial agents.

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. The inefficiency of *TVar* is also reflected in the fact that the *Lock-Based* implementation outperforms it on all cores. The sweet spot is in both cases at 3 cores, after which decreasing performance is the result. This is due to very similar approaches because both operate on the whole environment instead of only the cells as *TArray* does. In case of the *Lock-Based* approach, the lock contention increases, whereas in the *TVar* approach, the retries start to dominate (see Table 7).

Interestingly, the performance of the *TArray* implementation is the *worst* amongst all on 1 core. We attribute this to the overhead incurred by STM, which dramatically adds up in terms of a sequential execution.

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 can be seen in Figure 5b and which is also described in the book [20] section *Carrying Capacity* (p. 30).

We now measure the 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 [20]. 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

Agents	Sequential	Lock-Based	TVar	TArray
500	70.1 (0.41)	67.9 (0.13)	69.1 (0.34)	**25.7** (0.42)
1,000	145.0 (0.11)	130.0 (0.28)	136.0 (0.16)	**38.8** (1.43)
1,500	220.0 (0.14)	183.0 (0.83)	192.0 (0.73)	**40.1** (0.25)
2,000	213.0 (0.69)	181.0 (0.84)	214.0 (0.53)	**49.9** (0.82)
2,500	193.0 (0.16)	272.0 (0.81)	147.0 (0.32)	**55.2** (1.04)

Table 8: Performance comparison of *Sequential*, *Lock-Based*, *TVar* and *TArray* Sugarscape implementations with varying agent numbers and 50x50 environment on 4 cores (except *Sequential*). Timings in seconds (lower is better), standard deviation in parentheses.

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. 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 8 and plotted in Figure 7.

As expected, the *TArray* implementation outperforms all others substantially and scales up much smother. Also, *Lock-Based* performs better than the *TVar*.

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 payoff (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.

In case of the *Sequential* implementation with 2,000 agents we also arrive at a better performance than with 1,500, due to less space of the agents for free movement,

Cores	Carrying Capacity	Rebirthing
16	11.9 (0.21)	46.6 (0.07)
32	12.8 (0.29)	76.4 (0.01)
64	14.6 (0.09)	99.1 (0.01)

Table 9: Sugarscape *TArray* performance on 16, 32 and 64 cores an Amazon EC `m5ad.16xlarge` instance. Timings in seconds (lower is better). Retry ratios in parentheses.

exploiting laziness as in the case with 2,500 agents. In the case of the *Lock-Based* approach we see similar behaviour, where the performance with 2,000 agents is better than with 1,500. It is not quite clear why this is the case, given the dramatically *lower* performance with 2,500 agents but it seems that 2,000 agents create much less lock contention due to lower free space, whereas 2,500 agents create a lot more lock contention due to no free space available at all.

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 the *TArray* case, we ran the two experiments, carrying capacity (500 agents) and rebirthing (2500 agents), on an Amazon EC `m5ad.16xlarge` instance with 16, 32 and 64 cores to see if we run into decreasing returns. The results are reported in Table 9.

Unlike in the SIR model, Sugarscapes STM *TArray* implementation does not scale up beyond 16 cores. We attribute this to a mix of retries and Amdahl's law. As retries are much more expensive in the case of Sugarscape compared to SIR, even a small increase in the retry ratio (see Table 7), leads to reduced performance. On the other hand, although the retry ratio decreases as the number of cores increases, the ratio of parallelisable work diminishes and we get bound by the sequential part of the program.

5.5 Comparison with other approaches

The paper [38] reports a performance of 2,000 steps per second on a GPU on a 128x128 grid. Our best performing implementation, *TArray* with 500 rebirthing agents, arrives at a performance of 39 steps per second and is therefore clearly slower. However, 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 like 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 should be not as hard 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. This is the result of Amdahl's law where the environment becomes the limiting *sequential* 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 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 beyond the scope of this paper.

5.6 Summary

This case study showed clearly that besides being substantially faster than the *Sequential* implementation, an *STM* implementation with the right transactional data structure is also able to perform considerably better than a *Lock-Based* approach even in the case of the Sugarscape model which has a much higher complexity in terms of agent behaviour and dramatically increased number of writes to the environment.

Further, this case study demonstrated that the selection of the right transactional data structure is of fundamental importance when using STM. Selecting the right transactional data structure is highly model-specific and can lead to dramatically different performance results. In this case study 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. 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`.

Although the `TArray` is the better transactional data structure overall, it might come with an overhead, performing worse on low number of cores than a *TVar*, *Lock-Based* or even *Sequential* approach, as seen with *TArray* on 1 core. However, it has the benefit of quickly scaling up to multiple cores. 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.

The comparison between the *Lock-Based* approach and the *TArray* implementation seems to be a bit unfair due to a very different locking structure. A more suitable comparison would be to use an indexed Array with a tuple of `(MVar, IORef)`, holding a synchronisation primitive and reference for each cell to support fine-grained locking on the cell level. This would seem to be a more just comparison to the *TArray* where fine-grained transactions happen on the cell level. However, due to the model semantics, this approach is actually not possible. As already expressed in the experiments description, in Sugarscape an agent needs a consistent view of the whole environment for the whole duration of an agent execution due to the fact that agents don't only write their own locations but change also other locations. If we would use an indexed array we would also run into data races because the agents need to hold all relevant cells, which can't be grabbed in one atomic instruction but only one after another, which makes it highly susceptible to data races. Therefore, we could run into deadlocks if two agents are acquiring locks because they are taken after another and therefore subject to races where they end up holding a lock the other needs.

6 Conclusion

In this paper we investigated the potential of 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 implementations all consistently outperformed the lock-based ones and scaled much better to larger number of CPU cores. 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, files or network during simulation, the fact that no such interactions can occur within an agent when running within STM is not a problem.

Further, 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 [39, 40]. Also `TChan` offers a broadcast transactional channel, which supports broadcasting to listeners which maps nicely to a proactive 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 and suitable data structures, for example 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.

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 [38] but should be quite straightforward in STM using `TChan` and retries. However, we have yet to prove how to implement reliable synchronous agent interactions without deadlocks in STM. It might be very well the case that a truly concurrent approach is doomed due to the following [41] (Chapter 10. Software Transactional Memory, *What Can We Not Do with STM?*): *"In general, the class of operations that STM cannot express are those that involve multi-way communication between threads. The simplest example is a synchronous channel, in which both the reader and the writer must be present simultaneously for the operation to go ahead. We cannot implement this in STM, at least compositionally [..]: the operations need to block and have a visible effect — advertise that there is a blocked thread — simultaneously."*.

A drawback of STM is that it is not fair because *all* threads, which block on a transactional primitive, have to be woken up upon a change of the primitive, thus a FIFO guarantee cannot be given. We hypothesise that for most models, where the STM approach is applicable, this has no qualitative influence on the dynamics as agents are assumed to act conceptually at the same time and no fairness is needed. We leave the test of this hypothesis for future research. This is connected to our assumption that concurrent execution has no qualitative influence on the dynamics. Although repeated runs with same initial conditions might lead to different results due to nondeterminism, the dynamics follow still the same distribution as the one from the sequential implementation. To verify this we can make use the techniques of property-based testing as shown in [33] but we leave it for further research.

Declarations

Availability of data and materials

The datasets used and/or analysed during the current study are available from the corresponding author on reasonable request.

Competing interests

The authors declare that they have no competing interests.

Funding

Not applicable.

Authors' contributions

JT initiated the idea and the research, did the implementation, experiments, performance measurements, and writing. POS supervised the work, gave feedback and supported the writing process. All authors read and approved the final manuscript.

Acknowledgements

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

Authors' information

JONATHAN THALER is a Ph.D. student at the University of Nottingham and part of the Intelligent Modelling and Analysis Group (http://www.cs.nott.ac.uk/~psxjat/). His main research interest is the benefits and drawbacks of using pure functional programming with Haskell for implementing Agent-Based Simulations.

DR. PEER-OLAF SIEBERS is an Assistant Professor at the School of Computer Science, University of Nottingham, UK (http://www.cs.nott.ac.uk/~pszps/). His main research interest is the application of computer simulation to study human-centric complex adaptive systems. He is a strong advocate of Object Oriented Agent-Based Social Simulation. This is a novel and highly interdisciplinary research field, involving disciplines like Social Science, Economics, Psychology, Operations Research, Geography, and Computer Science. His current research focuses on Urban Sustainability and he is a co-investigator in several related projects and a member of the university's "Sustainable and Resilient Cities" Research Priority Area management team.

References

1. Macal CM. Everything you need to know about agent-based modelling and simulation. Journal of Simulation. 2016 May;10(2):144–156. Available from: https://link.springer.com/article/10.1057/jos.2016.7.
2. Suryanarayanan V, Theodoropoulos G, Lees M. PDES-MAS: Distributed Simulation of Multi-agent Systems. Procedia Computer Science. 2013 Jan;18:671–681. Available from: http://www.sciencedirect.com/science/article/pii/S1877050913003748.
3. Logan B, Theodoropoulos G. The distributed simulation of multiagent systems. Proceedings of the IEEE. 2001 Feb;89(2):174–185.
4. Lees M, Logan B, Theodoropoulos G. Using Access Patterns to Analyze the Performance of Optimistic Synchronization Algorithms in Simulations of MAS. Simulation. 2008 Oct;84(10-11):481–492. Available from: http://dx.doi.org/10.1177/0037549708096691.
5. Suryanarayanan V, Theodoropoulos G. Synchronised Range Queries in Distributed Simulations of Multiagent Systems. ACM Trans Model Comput Simul. 2013 Nov;23(4):25:1–25:25. Available from: http://doi.acm.org/10.1145/2517449.
6. Riley PF, Riley GF. 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; 2003. p. 817–825. Event-place: New Orleans, Louisiana. Available from: http://dl.acm.org/citation.cfm?id=1030818.1030926.

7. Gasser L, Kakugawa K. 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. New York, NY, USA: ACM; 2002. p. 745–752. Event-place: Bologna, Italy. Available from: http://doi.acm.org/10.1145/544862.544918.
8. Himmelspach J, Uhrmacher AM. Plug'n Simulate. In: 40th Annual Simulation Symposium (ANSS'07); 2007. p. 137–143. ISSN: 1080-241X.
9. Minson R, Theodoropoulos GK. Distributing RePast agent-based simulations with HLA. Concurrency and Computation: Practice and Experience. 2008;20(10):1225–1256. Available from: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1280.
10. Gorur BK, Imre K, Oguztuzun H, Yilmaz L. Repast HPC with Optimistic Time Management. In: Proceedings of the 24th High Performance Computing Symposium. HPC '16. San Diego, CA, USA: Society for Computer Simulation International; 2016. p. 4:1–4:9. Event-place: Pasadena, California. Available from: https://doi.org/10.22360/SpringSim.2016.HPC.046.
11. Hay J, Wilsey PA. 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. New York, NY, USA: ACM; 2015. p. 75–86. Event-place: London, United Kingdom. Available from: http://doi.acm.org/10.1145/2769458.2769462.
12. Abar S, Theodoropoulos GK, Lemarinier P, O'Hare GMP. Agent Based Modelling and Simulation tools: A review of the state-of-art software. Computer Science Review. 2017 May;24:13–33. Available from: https://linkinghub.elsevier.com/retrieve/pii/S1574013716301198.
13. Cicirelli F, Giordano A, Nigro L. Efficient Environment Management for Distributed Simulation of Large-scale Situated Multi-agent Systems. Concurr Comput : Pract Exper. 2015 Mar;27(3):610–632. Available from: http://dx.doi.org/10.1002/cpe.3254.
14. Hudak P, Hughes J, Peyton Jones S, Wadler P. A History of Haskell: Being Lazy with Class. In: Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages. HOPL III. New York, NY, USA: ACM; 2007. p. 12–1–12–55. Available from: http://doi.acm.org/10.1145/1238844.1238856.
15. Discolo A, Harris T, Marlow S, Jones SP, Singh S. Lock Free Data Structures Using STM in Haskell. In: Proceedings of the 8th International Conference on Functional and Logic Programming. FLOPS'06. Berlin, Heidelberg: Springer-Verlag; 2006. p. 65–80. Available from: http://dx.doi.org/10.1007/11737414_6.
16. Bezirgiannis N. Improving Performance of Simulation Software Using Haskells Concurrency & Parallelism; 2013.
17. Wilensky U, Rand W. An Introduction to Agent-Based Modeling: Modeling Natural, Social, and Engineered Complex Systems with NETLogo. MIT Press; 2015. Available from: https://www.amazon.co.uk/Introduction-Agent-Based-Modeling-Natural-Engineered/dp/0262731894.
18. Macal CM. To Agent-based Simulation from System Dynamics. In: Proceedings of the Winter Simulation Conference. WSC '10. Baltimore, Maryland: Winter Simulation Conference; 2010. p. 371–382. Available from: http://dl.acm.org/citation.cfm?id=2433508.2433551.
19. Thaler J, Altenkirch T, Siebers PO. Pure Functional Epidemics: An Agent-Based Approach. In: Proceedings of the 30th Symposium on Implementation and Application of Functional Languages. IFL 2018. New York, NY, USA: ACM; 2018. p. 1–12. Event-place: Lowell, MA, USA. Available from: http://doi.acm.org/10.1145/3310232.3310372.
20. Epstein JM, Axtell R. Growing Artificial Societies: Social Science from the Bottom Up. Washington, DC, USA: The Brookings Institution; 1996.
21. Shavit N, Touitou D. Software Transactional Memory. In: Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing. PODC '95. New York, NY, USA: ACM; 1995. p. 204–213. Available from: http://doi.acm.org/10.1145/224964.224987.
22. Perfumo C, Sönmez N, Stipic S, Unsal O, Cristal A, Harris T, et al. 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. New York, NY, USA: ACM; 2008. p. 67–78. Available from: http://doi.acm.org/10.1145/1366230.1366241.
23. Marlow S, Peyton Jones S, Singh S. Runtime Support for Multicore Haskell. In: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming. ICFP '09. New York, NY, USA: ACM; 2009.

p. 65–78. Available from: http://doi.acm.org/10.1145/1596550.1596563.

24. Harris T, Marlow S, Peyton-Jones S, Herlihy M. Composable Memory Transactions. In: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. PPoPP '05. New York, NY, USA: ACM; 2005. p. 48–60. Available from: http://doi.acm.org/10.1145/1065944.1065952.
25. Harris T, Peyton Jones S. Transactional memory with data invariants. In: First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT'06); 2006. Available from: https://www.microsoft.com/en-us/research/publication/transactional-memory-data-invariants/.
26. Thaler J, Siebers PO. The Art Of Iterating: Update-Strategies in Agent-Based. Springer Proceedings in Complexity. Dublin, Ireland: Springer International Publishing; 2017. Available from: https://www.springer.com/gp/book/9783030302979.
27. Kermack WO, McKendrick AG. A Contribution to the Mathematical Theory of Epidemics. Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences. 1927 Aug;115(772):700–721. Available from: http://rspa.royalsocietypublishing.org/content/115/772/700.
28. Enns RH. It's a Nonlinear World. 1st ed. Springer Publishing Company, Incorporated; 2010.
29. Meyer R. Event-Driven Multi-agent Simulation. In: Multi-Agent-Based Simulation XV. Lecture Notes in Computer Science. Springer, Cham; 2014. p. 3–16. Available from: https://link.springer.com/chapter/10.1007/978-3-319-14627-0_1.
30. Thaler J. Repository of STM implementations of the agent-based SIR model in Haskell; 2019. Last Access December 4, 2019. https://github.com/thalerjonathan/haskell-stm-sir. Available from: https://github.com/thalerjonathan/haskell-stm-sir.
31. libraries@haskell org. array: Mutable and immutable arrays; 2014. Last Access December 4, 2019. http://hackage.haskell.org/package/array. Available from: http://hackage.haskell.org/package/array.
32. van Dijk B, van Dijk J. concurrent-extra library; 2018. Last Access December 4, 2019. http://hackage.haskell.org/package/concurrent-extra. Available from: http://hackage.haskell.org/package/concurrent-extra.
33. Thaler J, Siebers PO. Show Me Your Properties! The Potential Of Property-Based Testing In Agent-Based Simulation. Berlin; 2019. .
34. O'Sullivan B. criterion: a Haskell microbenchmarking library; 2014. Last Access December 4, 2019. http://www.serpentine.com/criterion/. Available from: http://www.serpentine.com/criterion/.
35. O'Sullivan B. criterion: Robust, reliable performance measurement and analysis; 2014. Last Access December 4, 2019. http://hackage.haskell.org/package/criterion. Available from: http://hackage.haskell.org/package/criterion.
36. Breitner J. stm-stats library; 2019. Last Access December 4, 2019. http://hackage.haskell.org/package/stm-stats. Available from: http://hackage.haskell.org/package/stm-stats.
37. Thaler J. Repository of STM implementations of the Sugarscape model in Haskell; 2019. Last Access December 4, 2019. https://github.com/thalerjonathan/haskell-stm-sugarscape. Available from: https://github.com/thalerjonathan/haskell-stm-sugarscape.
38. Lysenko M, D'Souza RM. A Framework for Megascale Agent Based Model Simulations on Graphics Processing Units. Journal of Artificial Societies and Social Simulation. 2008;11(4):10. Available from: http://jasss.soc.surrey.ac.uk/11/4/10.html.
39. Agha G. Actors: A Model of Concurrent Computation in Distributed Systems. Cambridge, MA, USA: MIT Press; 1986.
40. Wooldridge M. An Introduction to MultiAgent Systems. 2nd ed. Wiley Publishing; 2009.
41. Marlow S. Parallel and Concurrent Programming in Haskell. O'Reilly; 2013. Google-Books-ID: k0W6AQAACAAJ.

Figures

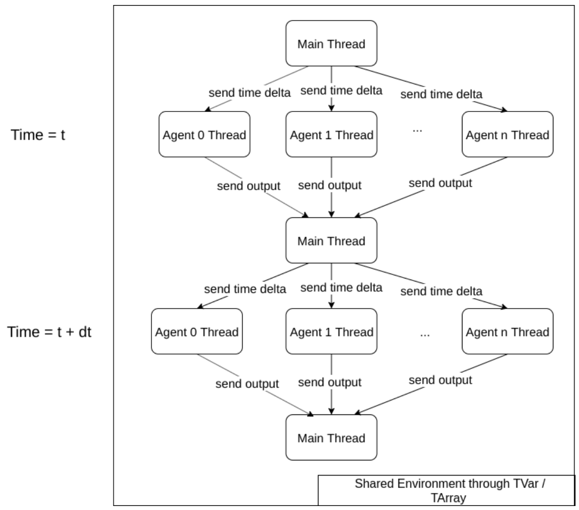


Figure 1: Diagram of the parallel time-driven lock-step approach.

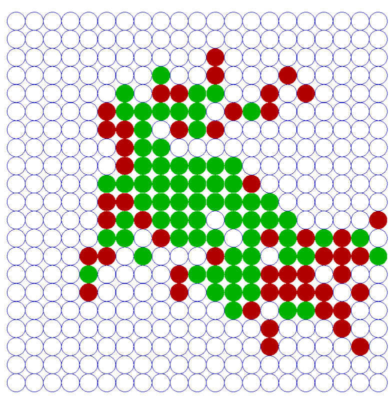


Figure 2: Simulation of the spatial SIR model with a Moore neighbourhood at $t = 100$. Initially a single infected agent at the center, contact rate $\beta = \frac{1}{5}$, infection probability $\gamma = 0.05$ and illness duration $\delta = 15$. Infected agents are indicated by red circles, recovered agents by green ones. The susceptible agents are rendered as blue hollow circles for better contrast.

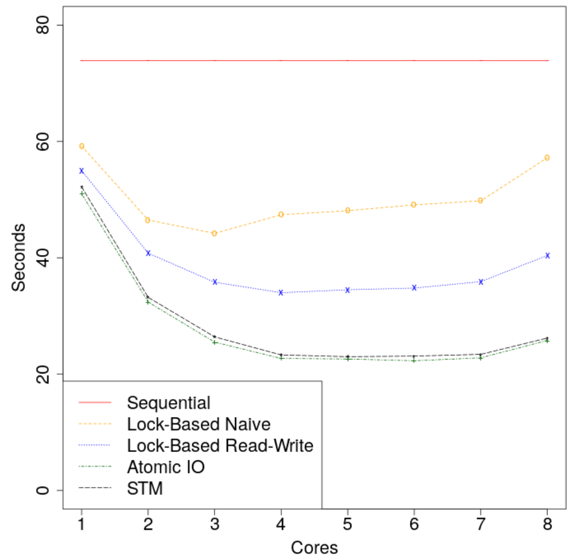


Figure 3: Performance comparison of *Sequential*, *STM*, *Lock-Based* and *Atomic IO* SIR implementations on varying cores with grid size of 51x51 (2,601) agents.

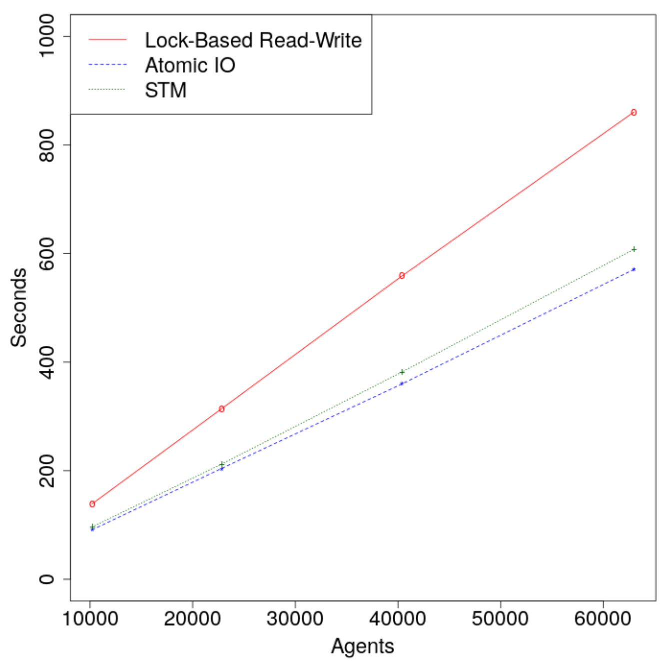
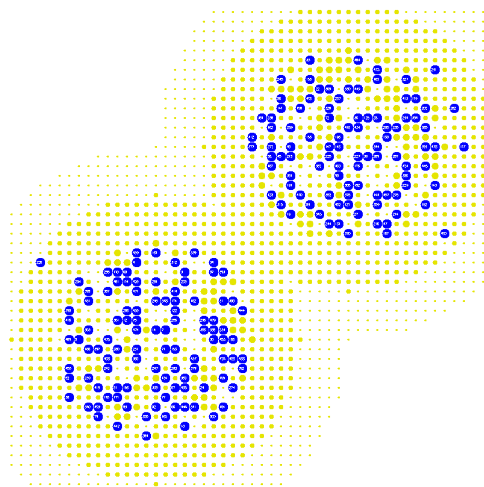
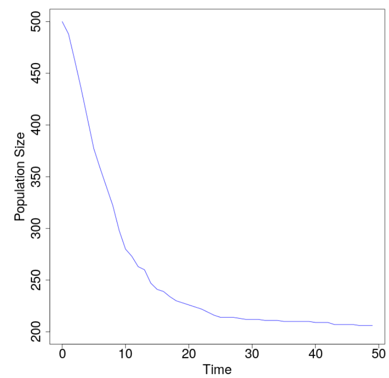


Figure 4: Performance comparison of *Lock-Based Read-Write*, *Atomic IO* and *STM* SIR implementations with varying grid sizes on 4 cores.



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



(b) Dynamics of the population size over 50 steps

Figure 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.

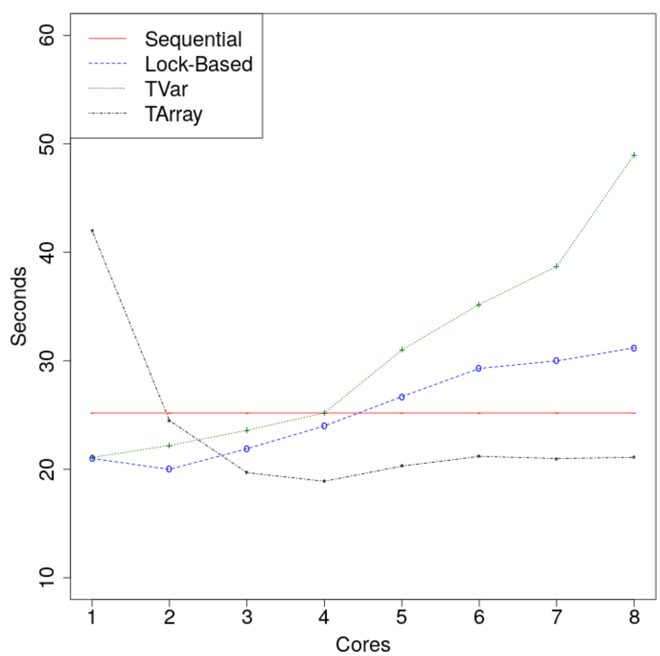


Figure 6: Performance comparison of *Sequential*, *Lock-Based*, *TVar* and *TArray* Sugarscape implementations on varying cores with 50x50 environment and 500 initial agents.

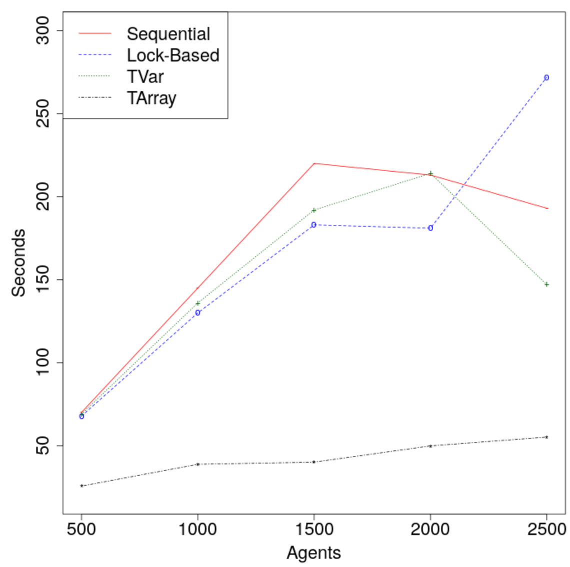


Figure 7: Performance comparison of *Sequential*, *Lock-Based*, *TVar* and *TArray* Sugarscape implementations with varying agent numbers and 50x50 environment on 4 cores (except *Sequential*).