A Tale Of Lock-Free Agents

The potential of Software Transactional Memory in parallel Agent-Based Simulation

JONATHAN THALER and THORSTEN ALTENKIRCH, 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. Unfortunately the established programming languages in the field, Python, Java and C++, are not very well suited to tackle the complexities of parallel programming. In this paper we propose the use of Software Transactional Memory (STM) in conjunction with the pure functional programming language Haskell which in combination allow to overcome the problems and complexities of lock-based parallel implementations of imperative approaches. We present two case studies where we compare the performance of lock-based and STM based implementations on 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 STM based 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.

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

ACM Reference Format:
Jonathan Thaler and Thorsten Altenkirch. 2019. A Tale Of Lock-Free Agents: The potential of Software Transactional Memory in parallel Agent-Based Simulation. 1, 1 (August 2019), 22 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The future of scientific computing in general and Agent-Based Simulation (ABS) in particular is parallelism because Moore's law is not holding any more as we have reached the physical limits of CPU clocks. The only escape is going massively parallel due to availability of cheap massive parallel local hardware with many cores or cloud services like Amazon S2. 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 not distinguish between data-parallelism and concurrency in the types.

Functional programming as in Haskell can provide a solution to both problems. The problems of lock-based approaches can be overcome by using Software Transactional Memory (STM). Although STM exists in other languages as well, Haskell was one of the first to natively build it into its core and the guaranteed lack of non-repeatable side-effects at compile time makes the use of STM in Haskell very compelling. Data-parallelism falls into place very naturally in Haskell because of the controlled side-effects and immutable data. The very unique benefit of using Haskell over existing STM implementations in other languages is that we can rule out any persistent side-effects in STM transactions which allows unproblematic retries of transactions - guaranteed at compile-time.

Authors' address: Jonathan Thaler, jonathan.thaler@nottingham.ac.uk; Thorsten Altenkirch, thorsten.altenkirch@nottingham.ac.uk, University of Nottingham, 7301 Wollaton Rd, Nottingham, NG8 1BB, United Kingdom.

2019. XXXX-XXXX/2019/8-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

We follow [7] and compare the performance of lock-based and STM implementations. also that paper gives a good indication how difficult and complex constructing a correct concurrent program is. the paper 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 hypothesise 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 [21] and the well known SugarScape [9] 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 empirically and experimentally investigate the benefit of using STM over lock-based approaches for concurrent ABS models. Although there exists research which has used STM in ABS [4], we explore it more rigorous and systematically on a conceptual level. Although we use the functional programming language Haskell and its STM implementation, we omit functional programming concepts almost altogether and focus only on Haskells ability to guarantee that transactions are truly repeatable without persistent side-effects which can be guaranteed at compile-time. Thus our contribution is that to the best of our knowledge we are the first to systematically investigate the use of STM in ABS and compare is performance with sequential, lock-based and imperative implementations both on local and Amazon Cloud Service machinery.

We start with Section 2 where we present related work and then present the concepts of STM in Section 3. In Section 4 we show how to apply STM to ABS in general and in our case-studies in particular. Section 5 contains the first case-study using a spatial SIR mode whereas Section 6 presents the second case-study using the SugarScape model. We conclude in Section 7 and give further research directions in Section 8.

2 RELATED WORK

In his masterthesis [4] the author investigated Haskells parallel and concurrency features to implement (amongst others) *HLogo*, a Haskell clone of the NetLogo 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 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, build on Functional Reactive Programming, explore the use of STM more on a more conceptual level rather than implementing a ABS library and compare our case-studies with lock-based and imperative implementations.

There exists some research [6, 31, 33] of using the functional programming language Erlang [3] to implement concurrent ABS. The language is inspired by the actor model [1] and was created in 1986 by Joe Armstrong for Eriksson for developing distributed high reliability software in telecommunications. The actor model can be seen as quite influential to the development of the concept of agents in ABS which borrowed it from Multi Agent Systems [37]. It emphasises message-passing concurrency with share-nothing semantics (no shared state between agents) which maps nicely to functional programming concepts. Erlang implements light-weight processes which allows to spawn thousands of them without heavy memory overhead. The mentioned papers investigate how the actor model can be used to close the conceptual gap between agent-specifications which focus on message-passing and their implementation. Further they also showed that using this kind of concurrency allows to overcome some problems of low level concurrent programming as well. Also [4] ported NetLogos API to Erlang mapping agents to concurrently running processes which interact with each other by message-passing. With some restrictions on the agent-interactions this model worked, which shows at using concurrent message-passing for parallel ABS is at least *conceptually* feasible.

The work [20] discusses a framework which allows to map Agent-Based Simulations to Graphics Processing Units (GPU). Amongst others they use the SugarScape model [9] and scale it up to millions of agents on very large environment grids. They reported an impressive speed-up of a factor of 9,000. Although their work is conceptually very different we can draw inspiration from their work in terms of performance measurement and comparison of the SugarScape model.

3 BACKGROUND

3.1 Functional Programming

Functional programming is called *functional* is because because it makes functions the main concept of programming, promoting them to first-class citizens. Its roots lie in the Lambda Calculus which was first described by Alonzo Church [5]. This is a fundamentally different approach to computation than imperative and object-oriented programming which roots lie in the Turing Machine [32]. Rather than describing *how* something is computed as in the more operational approach of the Turing Machine, due to the more declarative nature of the Lambda Calculus, code in functional programming describes *what* is computed.

In our research 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. The main points why we decided to go for Haskell are:

- Rich Feature-Set - it has all fundamental concepts of the pure functional programming paradigm of which we explain the most important below.
- Real-World applications - the strength of Haskell has been proven through a vast amount of highly diverse real-world applications [14], is applicable to a number of real-world problems [28] and has a large number of libraries available [1].
- Modern - Haskell is constantly evolving through its community and adapting to keep up with the fast changing field of computer science. Further, the community is the main source of high-quality libraries.

As a short example we give an implementation of the factorial function in Haskell:

```haskell
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n-1)
```

When looking at this function we can already see the central concepts of functional programming:

(1) Declarative - we describe *what* the factorial function is rather than how to compute it. This is supported by *pattern matching* which allows to give multiple equations for the same function, matching on its input.
(2) Immutable data - in functional programming we don't have mutable variables - after a variable is assigned, it cannot change its contents. This also means that there is no destructive assignment operator which can re-assign values to a variable. To change values, we employ recursion.

[1]https://wiki.haskell.org/Applications_and_libraries

(3) Recursion - the function calls itself with a smaller argument and will eventually reach the case of 0. Recursion is the very meat of functional programming because they are the only way to implement loops in this paradigm due to immutable data.
(4) Static Types - the first line indicates the name and the types of the function. In this case the function takes one Integer as input and returns an Integer as output. Types are static in Haskell which means that there can be no type-errors at run-time e.g. when one tries to cast one type into another because this is not supported by this kind of type-system.
(5) Explicit input and output - all data which are required and produced by the function have to be explicitly passed in and out of it. There exists no global mutable data whatsoever and data-flow is always explicit.
(6) Referential transparency - calling this function with the same argument will *always* lead to the same result, meaning one can replace this function by its value. This means that when implementing this function one can not read from a file or open a connection to a server. This is also known as *purity* and is indicated in Haskell in the types which means that it is also guaranteed by the compiler.

It may seem that one runs into efficiency-problems in Haskell when using algorithms which are implemented in imperative languages through mutable data which allows in-place update of memory. The seminal work of [27] showed that when approaching this problem from a functional mind-set this does not necessarily be the case. The author presents functional data structures which are asymptotically as efficient as the best imperative implementations and discusses the estimation of the complexity of lazy programs.

For an excellent and widely used introduction to programming in Haskell we refer to [16]. Other, more exhaustive books on learning Haskell are [2, 19]. For an introduction to programming with the Lambda-Calculus we refer to [25]. For more general discussion of functional programming we refer to [14, 15, 22].

3.1.1 *Side-Effects.*

One of the fundamental strengths of functional programming and Haskell is their 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 it 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): modifying a global variable, await an input from the keyboard, read or write to a file, open a connection to a server, drawing random-numbers,...

Obviously, to write real-world programs which interact with the outside-world we need side-effects. Haskell allows to indicate in the *type* of a function that it does or does *not* have side-effects. Further there are a broad range of different effect types available, to restrict the possible effects a function can have to only the required type. This is then ensured by the compiler which 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. Haskell also provides mechanisms to combine multiple effects e.g. one can define a function which can draw random-numbers and modify some global data. The most common side-effect types are:

- IO - 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,...
- Rand - Allows to draw random-numbers.
- Reader - Allows to read from an environment.
- Writer - Allows to write to an environment.
- State - Allows to read and write an environment.

A function with side-effects has to indicate this in their type e.g. if we want to give our factorial function for debugging purposes the ability to write to the standard output, we add IO to its type: factorial :: Integer -> IO Integer. 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. For example when running a function with a State-effect one needs to specify the initial environment which can be read and written. After running such a function with a State-effect the effect-runner returns the changed environment in addition with the return value of the function itself. 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. An effect-runner itself is a *pure* function. The exception to this is the IO effect type which does not have a runner but originates from the *main* function which is always of type IO.

Although it 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 errors already at compile-time which implies that we don't need test them with e.g. unit-tests. Second, because effect-runners are themselves *pure*, we can execute effectful functions in a very controlled way by making the effect-context explicit in the parameters to the effect-runner. This allows a much easier approach to isolated testing because the history of the system is made explicit.

For a technical, in-depth discussion of the concept of side-effects and how they are implemented in Haskell using Monads, we refer to the following papers: [17, 26, 34–36].

3.1.2 Parallelism and 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 but 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 IO or STM effect context.

Also, spawning thousands of threads in Haskell is no problem and has very low memory footprint because they are lightweight user-space threads, managed by the Haskell Runtime System which maps them to physical operating-system threads.

For a technical, in-depth discussion on parallelism and concurrency in Haskell we refer to the excellent book [23].

3.2 Software Transactional Memory

Software Transactional Memory (STM) was introduced by the paper [30] in 1995 as an alternative to lock-based synchronisation in concurrent programming which, in general, is notoriously difficult to get right because reasoning about the interactions of multiple concurrently running threads and low level operational details of synchronisation primitives and locks 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 utterly 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 depend on knowledge about their implementation. Also 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 very 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 threads have modified the same data as the STM actions thread, 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 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, this log is checked whether other threads have written to memory it has read - it checks whether it has a consistent view to the shared data or not. This might look like a serious overhead but the implementations are very mature by now, being very performant and the benefits outweigh its costs by far. In the paper [13] the authors used a model of STM to simulate optimistic and pessimistic STM behaviour under various scenarios using the AnyLogic simulation package. They concluded that optimistic STM may lead to 25% less retries of transactions. The authors of [29] 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.

3.3 STM in Haskell

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++,...) but it is in Haskell with its type-system and how side-effects are treated where it truly shines: 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.

Additionally, the following functions are provided:

- atomically :: STM a → IO a - Performs a series of STM actions atomically. Note that we need to run this in the IO Monad, which is obviously required when running an agent in a thread.
- retry :: STM a - Retry an action e.g. because a *TVar, TArray or TChan* (all build on TVar) does not contained required values. The runtime system blocks the action until one of the *TVars* has been updated.
- orElse :: STM a → STM a → STM a - Tries the first STM action and if it retries it will try the second one. If the second one retries as well, orElse as a whole retries.
- check :: Bool → STM () - If the given boolean condition is False then the action will retry. Allows to encode invariants explicitly in code.

4 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. See Figure 1 for a visualisation of our 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 sesctions, 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 visualisation purposes, the main thread can access the *TVar* and *TArray* as well.

5 CASE STUDY 1: SPATIAL SIR

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 [8].

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 [21] but extended it by placing the agents on a discrete 2D grid using a Moore (8) neighbourhood TODO: cite my own PFE paper. In this case agents interact with each other indirectly through the shared discrete 2D grid by writing their current state on their cell which neighbours can read. 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 [24] approach and maps naturally to the continuous time-semantics

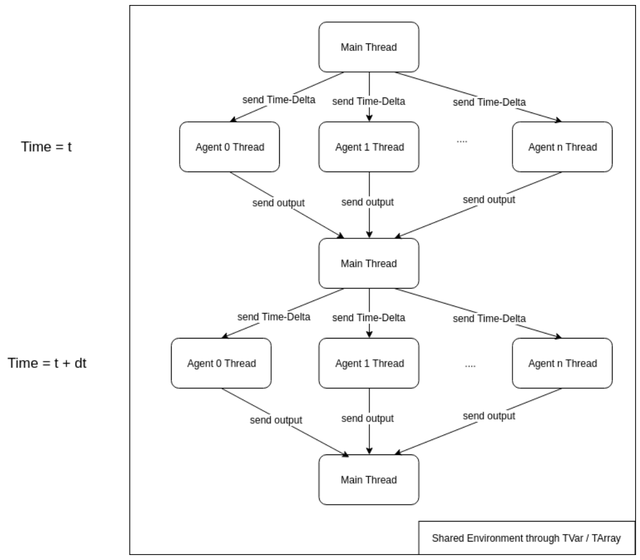


Fig. 1. Diagram of the parallel time-driven lock-step approach.

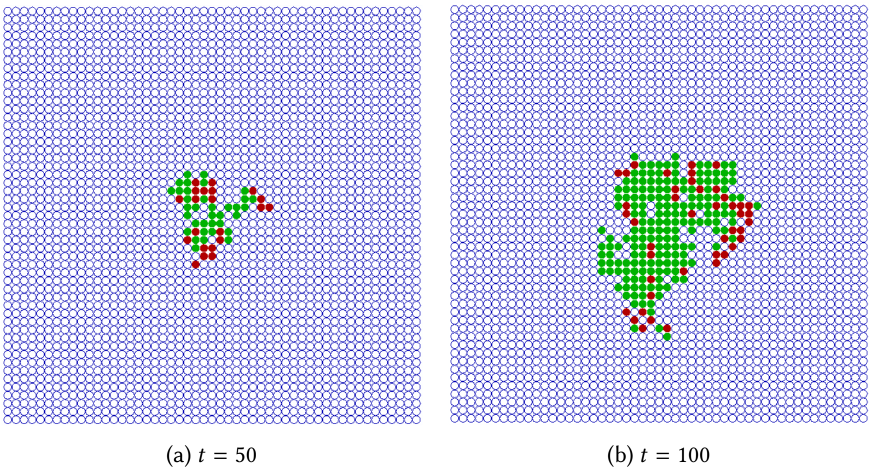


Fig. 2. Simulating the agent-based SIR model on a 51x51 2D grid with Moore neighbourhood, a single infected agent at the center, contact rate $\beta = \frac{1}{5}$, infection probability $\gamma = 0.05$ and illness duration $\delta = 15$. Simulation run until $t = 100$ with fixed $\Delta t = 0.1$. The susceptible agents are rendered as blue hollow circles for better contrast.

and state-transitions provided by FRP. By sampling the system with very small Δt this means that we have comparatively very few writes to the shared environment which will become important when discussing the performance results.

OS	Fedora 28 64-bit
RAM	16 GByte
CPU	Intel Core i5-4670K @ 3.40GHz x 4
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

5.1 Experiment Design

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

(1) Sequential - This is the original implementation we also discuss in TODO: cite my own PFE paper. In it the discrete 2D grid is shared amongst all agents using the State Monad. Agents are run sequentially after another thus ensuring exclusive read/write access to it. Because we are neither running in the STM or IO Monad there is no way we can run this implementation concurrently.

(2) STM - This is the same implementation like the State Monad but instead of sharing the discrete 2D grid in a State Monad, agents run in the STM Monad and have access to the discrete 2D grid through a transactional variable *TVar*. This means that the reads and writes of the discrete 2D grid are exactly the same but happen always through the *TVar*. Also each agent is run within its own thread, thus enabling true concurrency when the simulation is actually run on multiple cores (which can be configured by the Haskell Runtime System).

(3) Lock-Based - This is exactly the same implementation like the STM Monad but instead of running in STM, the agents now run in IO. They share the discrete 2D grid using an *IORef* and have access to an *MVar* to synchronise access to the it. Also each agent is run within its own thread.

(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 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. Also 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 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 average. Further, we checked the visual outputs and the dynamics and they look qualitatively the same to the reference implementation of the State Monad TODO: cite my own PFE paper. In the experiments we varied the number of agents (grid size) and the number of cores when running concurrently - the numbers are always indicated clearly. For varying the number of cores we compiled the executable using *stack* and the *threaded* option and executed it with *stack* using the +RTS -Nx option where x is the number of cores between 1 and 4.

	Cores	Duration
Sequential	1	100.3
STM	1	53.2
STM	2	27.8
STM	3	21.8
STM	4	20.2
Lock-Based	1	60.6
Lock-Based	2	42.8
Lock-Based	3	38.6
Lock-Based	4	41.6
RePast	1	**10.822**

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

5.2 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 on multiple cores of the STM and Lock-Based implementations shows that the lock-free 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 seen 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 more potential for reduced performance. This is not an issue in STM because no locks are taken in advance.

Comparing the reference *State* implementation shows that it is the slowest by far - even the single core STM and Lock-Based implementations outperform it by far. Also our profiling results reported about 30% increased memory footprint for the State implementation. This shows that the State Monad is a rather slow and memory intense approach sharing data but guarantees purity and excludes any non-deterministic side-effects which is not the case in STM and IO.

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). We attribute this to the conceptually slower approach of functional programming. We might could have optimised parts of the code but leave this for further research.

5.3 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. The results for STM are reported in Table 3. Again, 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.

We plotted the results in Figure 4. It is clear that the lock-free STM implementation outperforms the lock-based 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

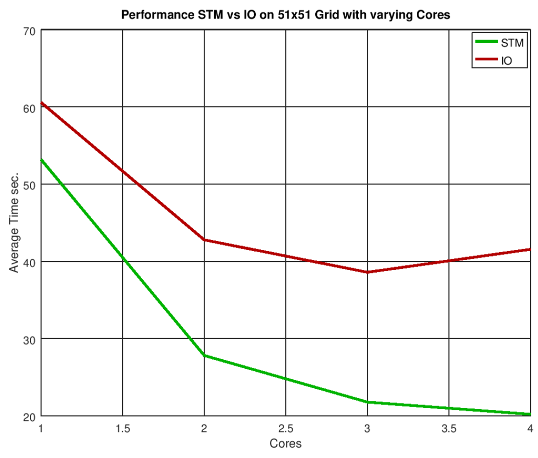


Fig. 3. Comparison of performance and scaling on multiple cores of STM vs. IO. Note that the Lock-Based implementation performs worse on 4 cores than on 3.

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	464.017
201 x 201 (40,401)	**302.4**	672.0	720.6	1,227.68
251 x 251 (63,001)	**495.7**	1,027.3	1,117.2	3,283.63

Table 3. Performance on varying grid sizes.

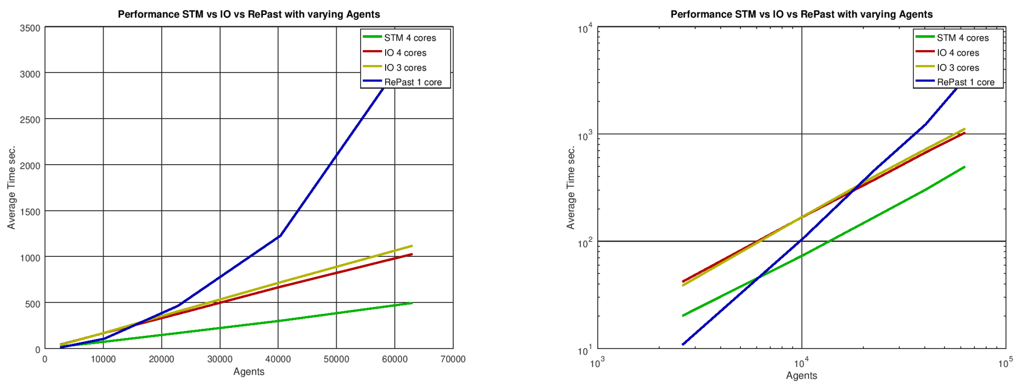
Grid-Size	Commits	Retries	Ratio
51 x 51 (2,601)	2,601,000	1306.5	0.0
101 x 101 (10,201)	10,201,000	3712.5	0.0
151 x 151 (22,801)	22,801,000	8189.5	0.0
201 x 201 (40,401)	40,401,000	13285	0.0
251 x 251 (63,001)	63,001,000	21217	0.0

Table 4. Retries Ratio of STM Monad experiments on varying grid sizes on 4 cores.

of 2. The figure is inverted on a 251x251 grid where the 4 core Haskell STM version outperforms 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 never have anticipated an increase of factor 6.

5.4 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. In these experiments we only averaged over 4 runs because they all arrived at a ratio of 0.0. The results are reported in Table 4.



(a) Normal Scale

(b) Logarithmic scale on both axes

Fig. 4. Performance on varying grid sizes.

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

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

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 substantial better performance over the Lock-Based implementation. Obviously this ratio stems from the fact, that in our implementation we have *very* few writes (only when an agent changes e.g. from Susceptible to Infected or from Infected to Recovered) and mostly reads. Also we conducted runs on lower number of cores which resulted in fewer retries, which was what we expected.

5.5 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 the two experiments (51x51 and 251x251) on Amazon S2 instances with increasing number of cores starting with 16 until we ran 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 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. TODO: why? need to check retries TODO: the INCREASE in time can only happen due to more retries

5.6 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- and memory-efficient than running in the State Monad 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 *for that model.*
(3) STM outperforms the Lock-Based implementation substantially and scales much better to multiple cores.
(4) STM on single (1) core is still about twice as slow than 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.

6 CASE STUDY 2: SUGARSCAPE

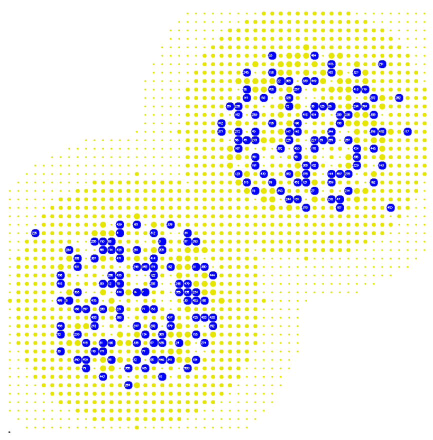
One of the first models in Agent-Based Simulation was the seminal Sugarscape model developed by Epstein and Axtell in 1996 [9]. Their aim was to *grow* an artificial society by simulation and connect observations in their simulation to phenomenon observed in real-world societies. In this model a population 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 [9]. There, in each step agents search (move) to the cell with the highest sugar they see within 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:

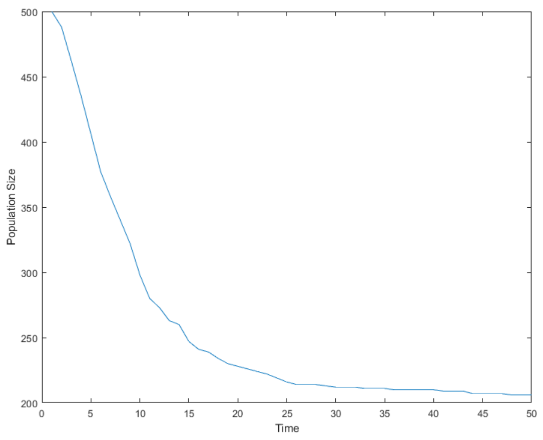
- Sugar Endowment: each agent has an initial sugar endowment randomly uniform distributed between 5 and 25 units.
- Sugar Metabolism: each agent has a sugar metabolism randomly uniform distributed between 1 and 5.
- Agent Vision: each agent has a vision randomly uniform distributed between 1 and 6, same 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.

6.1 Experiment Design

We compare three different implementations



(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.

(1) Sequential - All agents are run after another (including the environment) and the environment is shared amongst the agents using the State Monad.
(2) Lock-Based - All agents are run concurrently and the environment is shared using an *IORef* 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). In the *Sequential* approach we do this explicitly but in both STM approaches this happens automatically due to race-conditions in concurrency thus we arrive at an effectively shuffled processing of agents: we can assume that the order of the agents is *effectively* random in every step. The important difference between the two approaches is that in the State 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 could have two options for running the environment: either running it 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. It could happen 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 (TODO: reference the book where it shows that environment grows after / before all agents).

We follow [20] and measure the average updates 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. For varying the number of cores we

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

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

compiled the executable using *stack* and the *threaded* option and executed it with *stack* using the *+RTS -Nx* option where x is the number of cores between 1 and 4.

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 crucial 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. 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 ridiculous high number of steps per second but which obviously don't really reflect the true computations done.

6.2 Constant Agent Size

In this first approach we compare the performance of all implementations on varying numbers of cores. The results are reported in Table 6 and can be seen 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 read 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 *TArray* seems to scale up by 10 steps per second for every core added, it would be interesting to see how far this could go as we seem not to hit a limit with 4 cores yet. We leave this for further research.

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 cores which then drops on 4 cores which the *Lock-Based* approach seems to be able to avoid but reducing its returns on increased number of cores hitting a limit there as well.

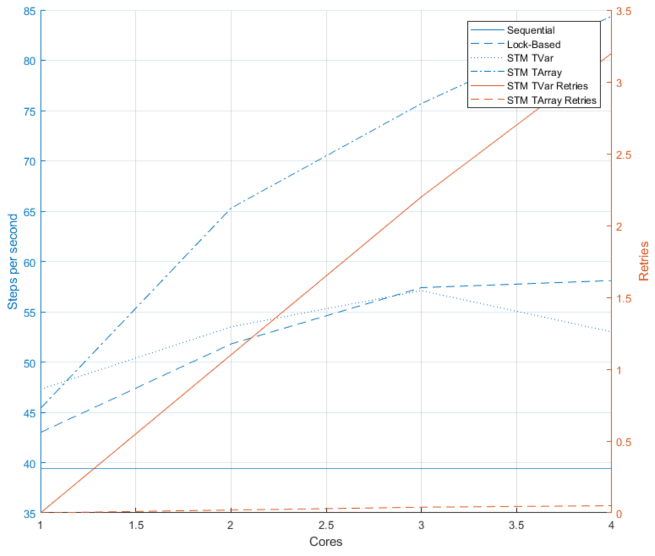


Fig. 6. Steps per second and retries on 50x50 grid and 500 initial agents on varying cores.

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 and varying number of agents with 4 (and 3) cores except Sequential (1 core).

6.3 Scaling up Agents

So far we always 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 [9] 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 [9]. 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 (3) 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 7 and can be seen 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

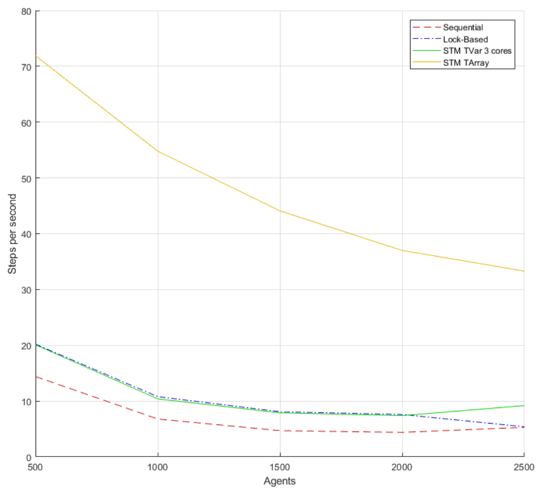


Fig. 7. Steps per second on 50x50 grid and varying number of agents with 4 (and 3) cores except Sequential (1 core).

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, when an agent tries to move it 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. 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 we perform STM reads on the neighbouring cells which are not subject to lazy evaluation.

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.

6.4 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 until we ran into decreasing returns. The results are reported in Table 8.

	Cores	Carrying Capacity	Rebirthing
Lock-Based	16	53.9	4.4
Lock-Based	32	44.2	3.6
STM TArray	16	116.8	39.5
STM TArray	32	109.8	31.3

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

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. At this point we are running into Amdahls law and become dominated by the sequential part of the program which is the main lock-step mechanism and environment process. TODO: the INCREASE in time can only happen due to more retries

6.5 Comparison with other approaches

The paper [20] reports a performance of 17 steps in RePast, 18 steps in MASON (both non-parallel) and 2000 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 one must be very well aware that these results were generated in 2008, on 10 year older hardware - the performance might have caught up by now and even outperform our functional *Sequential* approach.

Indeed, 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 more than 5 faster than even our STM *TArray* implementation on 4 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 deeper complexity of the model where it seems that imperative implementations seem to have an advantage. Still our research is just a first step and might result in future work increasing performance.

The very high performance on the GPU does not concern us here as it follows a very different approach than we do here. Our focus is 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 approach. 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 feature as bilateral trading between agents where a pair of agents need to come to a conclusion over multiple synchronous steps is difficult or even impossible to implement on a GPU whereas this is easily possible using STM on a CPU as well.

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 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.

6.6 Discussion

Reflecting of 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, 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 earlier or later. In the case of the *TVar* the best performance is reached with 3 cores. With the *TArray* we didn't reach this limit yet with 4 cores - we leave this for further research as it might be well beyond tens of 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.
- Generalise the insight of 2,500 better than 2,000
- Unfortunately for this model the performance is nowhere comparable to imperative approaches which we attribute to the inherent deeper complexity of the model where it seems that imperative implementations seem to have an advantage.

7 CONCLUSION

In this paper we investigated the potential for using STM for concurrent, large-scale ABS and come to the conclusion that it is a very promising approach and alternative over lock-based approaches as our proof-of-concept has 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.

Interestingly, STM primitives map nicely to ABS concepts. Using a shared environment either using *TVar* or *TArray*, depending on its nature. Also there exists the *TChan* primitive which can be seen as persistent message boxes for agents, underlining the message-oriented approach found in many agent-based models. 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 concurrency operational details.

The strong static type-system of Haskell adds another benefit. By running in STM instead of IO 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 as we have shown.

After the strong performance results of the SIR case-study in Section 5 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 although functional program has caught up in speed, it is still behind imperative approaches. Also to squeeze out high performance of functional programs which can catch up with imperative implementations involves much more experience and sophisticated techniques than just writing imperative approaches. Also the Amazon experiments suggest that we are running into a limit between 16 and 32 cores on the STM implementation which is most probably due to the synchronous lock-step approach. There are a number of techniques which could allow the speeding up

Despite the indisputable benefits of using STM within a pure functional setting like Haskell, it exists also in other imperative languages (TODO: cite, enumerate for python, Java and C++). 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.

8 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 [20] but should be quite straightforward in STM using *TChan* and retries. We leave this for future research.

We have not focused on implementing an approach like *Sense-Think-Act* cycle as mentioned in [38]. 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 but leave this for future research.

So far we only looked at a time-driven models (note that the Sugarscape is basically a time-driven model where each agent acts in each step) where all agents are run concurrently in lock-step. 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 should be theoretically be implemented using Parallel Discrete Event Simulation (PDES) [10] and it would be interesting to see how an event-driven ABS approach based on an underlying PDES implementation would perform.

In our Sugarscape implementation we didn't scale up the environment because it is the limiting factor due to its sequential nature. Partitioning the environment into subsets which can be updated concurrently / parallel could speed up the environment updating as well which should be particularly easy in functional programming and using STM *TArray*. We leave this for future research.

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

ACKNOWLEDGMENTS

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

REFERENCES

[1] Gul Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA.
[2] Christopher Allen and Julie Moronuki. 2016. *Haskell Programming from First Principles.* Allen and Moronuki Publishing. Google-Books-ID: 5FaXDAEACAAJ.
[3] Joe Armstrong. 2010. Erlang. *Commun. ACM* 53, 9 (Sept. 2010), 68–75. https://doi.org/10.1145/1810891.1810910
[4] Nikolaos Bezirgiannis. 2013. *Improving Performance of Simulation Software Using Haskells Concurrency & Parallelism.* Ph.D. Dissertation. Utrecht University - Dept. of Information and Computing Sciences.
[5] Alonzo Church. 1936. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics* 58, 2 (April 1936), 345–363. https://doi.org/10.2307/2371045
[6] Antonella Di Stefano and Corrado Santoro. 2005. Using the Erlang Language for Multi-Agent Systems Implementation. In *Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT '05).* IEEE Computer Society, Washington, DC, USA, 679–685. https://doi.org/10.1109/IAT.2005.141
[7] 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
[8] Richard H. Enns. 2010. *It's a Nonlinear World* (1st ed.). Springer Publishing Company, Incorporated.
[9] Joshua M. Epstein and Robert Axtell. 1996. *Growing Artificial Societies: Social Science from the Bottom Up.* The Brookings Institution, Washington, DC, USA.
[10] Richard M. Fujimoto. 1990. Parallel Discrete Event Simulation. *Commun. ACM* 33, 10 (Oct. 1990), 30–53. https://doi.org/10.1145/84537.84545
[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] 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
[14] 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
[15] J. Hughes. 1989. Why Functional Programming Matters. *Comput. J.* 32, 2 (April 1989), 98–107. https://doi.org/10.1093/comjnl/32.2.98
[16] Graham Hutton. 2016. *Programming in Haskell.* Cambridge University Press. Google-Books-ID: 1xHPDAAAQBAJ.
[17] Simon Peyton Jones. 2002. Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering theories of software construction.* Press, 47–96.
[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] Miran Lipovaca. 2011. *Learn You a Haskell for Great Good!: A Beginner's Guide* (1 edition ed.). No Starch Press, San Francisco, CA.
[20] 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
[21] 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
[22] Bruce J. MacLennan. 1990. *Functional Programming: Practice and Theory.* Addison-Wesley. Google-Books-ID: JqhQAAAAMAAJ.
[23] Simon Marlow. 2013. *Parallel and Concurrent Programming in Haskell.* O'Reilly. Google-Books-ID: k0W6AQAACAAJ.

[24] 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

[25] Greg Michaelson. 2011. *An Introduction to Functional Programming Through Lambda Calculus*. Courier Corporation. Google-Books-ID: gKvwPtvsSjsC.

[26] E. Moggi. 1989. Computational Lambda-calculus and Monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*. IEEE Press, Piscataway, NJ, USA, 14–23. http://dl.acm.org/citation.cfm?id=77350.77353

[27] Chris Okasaki. 1999. *Purely Functional Data Structures*. Cambridge University Press, New York, NY, USA.

[28] Bryan O'Sullivan, John Goerzen, and Don Stewart. 2008. *Real World Haskell* (1st ed.). O'Reilly Media, Inc.

[29] Cristian Perfumo, Nehir SÃűnmez, Srdjan Stipic, Osman Unsal, AdriÃạn 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

[30] 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

[31] Gene I. Sher. 2013. *Agent-Based Modeling Using Erlang Eliminating The Conceptual Gap Between The Programming Language & ABM*.

[32] A. M. Turing. 1937. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society* s2-42, 1 (1937), 230–265. https://doi.org/10.1112/plms/s2-42.1.230

[33] Carlos Varela, Carlos Abalde, Laura Castro, and Jose GulÃŋas. 2004. On Modelling Agent Systems with Erlang. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Erlang (ERLANG '04)*. ACM, New York, NY, USA, 65–70. https://doi.org/10.1145/1022471.1022481

[34] Philip Wadler. 1992. The Essence of Functional Programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '92)*. ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/143165.143169

[35] Philip Wadler. 1995. Monads for Functional Programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*. Springer-Verlag, London, UK, UK, 24–52. http://dl.acm.org/citation.cfm?id=647698.734146

[36] Philip Wadler. 1997. How to Declare an Imperative. *ACM Comput. Surv.* 29, 3 (Sept. 1997), 240–263. https://doi.org/10.1145/262009.262011

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

[38] 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 May 2018