A Tale Of Lock-Free Agents

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

JONATHAN THALER and THORSTEN ALTENKIRCH, University of Nottingham, United Kingdom

TODO: select journals - ACM Transactions on Modeling and Computer Simulation (TOMACS): https://tomacs.acm.org/ - ?

TODO: do 8 runs for all SIR experiments, some are lacking: runs where STD (σ Duration) is 0 had no full 8 runs yet

TODO: sugarscape case-study - do profiling and look into where the performance is lost - implement IO version if not too much effort, maybe use Data.Array.IO so we can directly re-use Discrete 2D - run measurements on my machine

TODO: write performance discussion TODO: write the background section TODO: write STM and ABS TODO: write Introduction TODO: write conclusion TODO: write further Research

TODO: what about box-blots? we have standard-deviation for each data-point so we could give ranges?

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

ACM Reference Format:
Jonathan Thaler and Thorsten Altenkirch. 2019. A Tale Of Lock-Free Agents: The potential of Software Transactional Memory in concurrent Agent-Based Simulation. 1, 1 (July 2019), 17 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION (ONCE UPON A TIME...)

In the paper [11] 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.

why FP? because concurrency and parallelism in general more easier in FP due to controlled side-effects and immutable data. also strong benefit is that STM is built into the language based on leightweight thread system. unique benefit is that we can rule out any persistent side-effects in STM transactions which allows unproblematic retries of transactions - guaranteed at compile-time

We follow [6] and compare the Performance of lock based and lock free 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 hope the same results for our paper

We present case-studies in which we employ the well known SugarScape [8] and agent-based spatial SIR [14] model to test our hypothesis. The former 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 latter 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.

problem of low-level lock-based concurrency programming - inefficiency e.g. more contention for aquiring a lock - complexity e.g. forgetting to releasing a lock or re-taking it can lead to deadlocks. in complex programs this is not obviously detectable

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/2019/7-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The aim of this paper is to empirically and experimentally investigate the benefit of using STM for concurrent ABS models. Although there exists research which has used STM in ABS [3], 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.

This paper makes the following contributions: - STM to implement concurrent ABS - compares 3 approaches: non-concurrent, low-level locking, STM

The structure of the paper is:

2 BACKGROUND

TODO: The Limits of Software Transactional Memory [16] TODO: Composable Memory Transactions [9] TODO: Transactional memory with data invariants [10] TODO: A survey on parallel and distributed multi-agent systems for high performance computing simulations [17] TODO: Software Transactional Memory vs. Locking in a Functional Language [4]

TODO: be very careful, i copied some sentences directly from the relevant papers The whole concept of our approach is built on the usage of Software Transactional Memory (STM), where we follow the main paper [9] on STM [1].

Concurrent programming 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 wakeups 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. In STM one executes actions atomically where modifications made in such an action are invisible to other threads until the action is performed. Also the thread in which this action is run, doesn't see changes made by other threads - thus execution of STM actions are isolated. When a transaction exits one of the following things will occur:

(1) If no other thread concurrently modified the same data as us, all of our modifications will simultaneously become visible to other threads.
(2) Otherwise, our modifications are discarded without being performed, and our block of actions is automatically restarted.

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

[1]We also make use of the excellent tutorial http://book.realworldhaskell.org/read/software-transactional-memory.html.

STM is implemented using optimistic synchronisation. This 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.

Applying this to our agents is very simple: because we already use Dunai / BearRiver as our FRP library, we can run in arbitrary Monadic contexts. This allows us to simply run agents within an STM Monad and execute each agent in their own thread. This allows then the agents to communicate concurrently with each other using the STM primitives without problems of explicit concurrency, making the concurrent nature of an implementation very transparent. Further through optimistic synchronisation we should arrive at a much better performance than with low level locking.

2.1 STM primitives

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 - Allows to retry a transaction immediately.
- 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.

3 RELATED WORK

In his masterthesis [3] the author investigated Haskells parallel and concurrency features to implement (amongst others) *HLogo*, a Haskell clone of the NetLogo simulation package, focusing on using Software Transactional Memory 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 FRP and explore on a more conceptual level the use of STM rather than implementing a ABS library.

There exists some research [5, 18, 19] of using the functional programming language Erlang [2] 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 [20]. 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 [3] 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 [13] discusses a framework which allows to map Agent-Based Simulations to Graphics Processing Units (GPU). Amongst others they use the SugarScape model [8] 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.

TODO:

4 STM AND ABS

For a proof-of-concept we changed the reference implementation of the agent-based SIR model on a 2D-grid as described in the paper in Appendix **??**. In it, a State Monad is used to share the grid across all agents where all agents are run after each other to guarantee exclusive access to the state. We replaced the State Monad by the STM Monad, share the grid through a *TVar* and run every agent within its own thread. All agents are run at the same time but synchronise after each time-step which is done through the main-thread.

We make STM the innermost Monad within a RandT transformer:

```haskell
type SIRMonad g   = RandT g STM
type SIRAgent g   = SF (SIRMonad g) () ()
```

In each step we use an *MVar* to let the agents block on the next Δt and let the main-thread block for all results. After each step we output the environment by reading it from the *TVar*:

```haskell
-- this is run in the main-thread
simulationStep :: TVar SIREnv
               -> [MVar DTime]
               -> [MVar ()]
               -> Int
               -> IO SIREnv
simulationStep env dtVars retVars _i = do
  -- tell all threads to continue with the corresponding DTime
  mapM_ (`putMVar` dt) dtVars
  -- wait for results, ignoring them, only [()]
  mapM_ takeMVar retVars
  -- read last version of environment
  readTVarIO env
```

Each agent runs within its own thread. It will block for the posting of the next Δt where it then will run the MSF stack with the given Δt and atomically transacting the STM action. It will then post the result of the computation to the main-thread to signal it has finished. Note that the number of steps the agent will run is hard-coded and comes from the main-thread so that no infinite blocking occurs and the thread shuts down gracefully.

```haskell
createAgentThread :: RandomGen g
                  => Int
                  -> TVar SIREnv
                  -> MVar DTime
                  -> g
                  -> (Disc2dCoord, SIRState)
```

```
                   -> IO (MVar ())
createAgentThread steps env dtVar rng0 a = do
    let sf = uncurry (sirAgent env) a
    -- create the var where the result will be posted to
    retVar <- newEmptyMVar
    _ <- forkIO (sirAgentThreadAux steps sf rng0 retVar)
    return retVar
  where
    agentThread :: RandomGen g
                => Int
                -> SIRAgent g
                -> g
                -> MVar ()
                -> IO ()
    agentThread 0 _ _ _ = return ()
    agentThread n sf rng retVar = do
      -- wait for next dt to compute next step
      dt <- takeMVar dtVar

      -- compute next step
      let sfReader = unMSF sf ()
          sfRand   = runReaderT sfReader dt
          sfSTM    = runRandT sfRand rng
      ((_, sf'), rng') <- atomically sfSTM

      -- post result to main thread
      putMVar retVar ()

      agentThread (n - 1) sf' rng' retVar
```

5 CASE STUDY 1 - SPATIAL SIR (FIRST ENCOUNTER)

Our first case study is the SIR model which is a very well studied and understood compartment model from epidemiology [12] which allows to simulate the dynamics of an infectious disease like influenza, tuberculosis, chicken pox, rubella and measles spreading through a population [7].

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 [14] 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 1.

It is important to note that due to the continuous-time nature of the SIR model, our implementation follows the time-driven [15] approach and maps naturally to the continuous time-semantics 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.

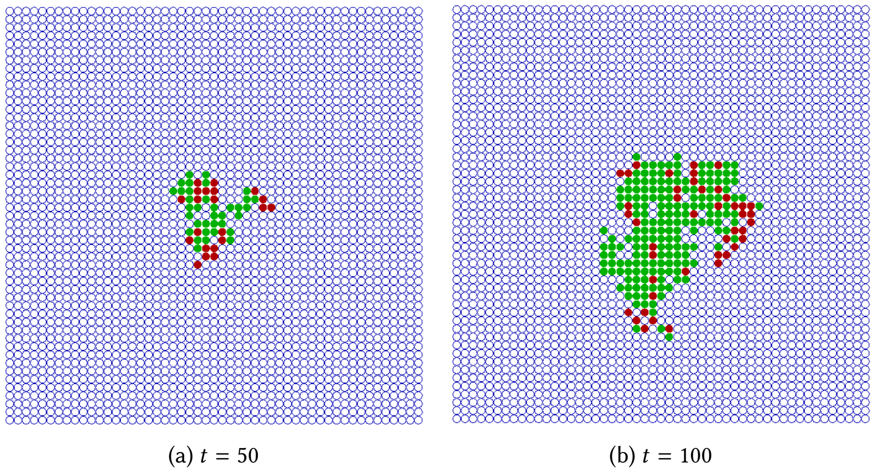


Fig. 1. Simulating the agent-based SIR model on a 51x51 2D grid with Moore neighbourhood, a single infected agent at the center, contact rate $\beta = \frac{1}{5}$, infection probability $\gamma = 0.05$ and illness duration $\delta = 15$. Simulation run until $t = 100$ with fixed $\Delta t = 0.1$. The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

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) State - 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 IO - 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.

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

	Cores	μ Duration (sec)	σ Duration (sec)
State	1	100.33	0.434
STM	1	53.182	0.393
STM	2	27.817	0.555
STM	3	21.776	0.388
STM	4	20.201	0.789
IO	1	60.564	0.265
IO	2	42.779	0.421
IO	3	38.586	0.451
IO	4	41.555	0.445
RePast	1	**10.822**	0.377

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

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 both the average and standard deviation. 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.

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 IO implementations shows that the lock-free STM implementation significantly outperforms the lock-based IO one and scales better to multiple cores. The IO implementation performs best with 3 cores and shows slightly worse performance on 4 cores as can be seen in Figure 2. 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 IO implementations outperform it by far. Also our profiling results reported

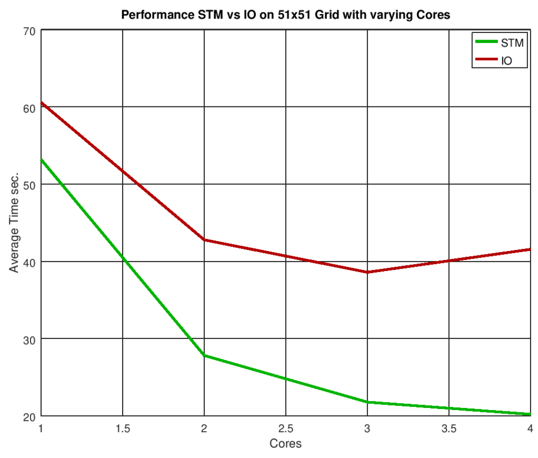


Fig. 2. Comparison of performance and scaling on multiple cores of STM vs. IO. Note that the IO implementation performs worse on 4 cores than on 3.

Grid-Size	μ Duration (sec)	σ Duration (sec)
51 x 51 (2,601)	20.201	0.789
101 x 101 (1,0201)	**74.493**	0.524
151 x 151 (22,801)	**168.47**	1.783
201 x 201 (40,401)	**302.43**	3.931
251 x 251 (63,001)	**495.73**	0

Table 3. STM Monad experiments on varying grid sizes on 4 cores.

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, IO performed best on 3 cores, we additionally ran IO on 3 cores as well. The results for STM are reported in Table 3, for IO in Tables 4, 5 and Repast in Table 6. 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 3. It is clear that the lock-free STM implementation outperforms the lock-based IO implementation by a substantial factor. Surprisingly, the IO 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

Grid-Size	μ Duration (sec)	σ Duration (sec)
51 x 51 (2,601)	41.914	1.073
101 x 101 (10,201)	170.55	1.115
151 x 151 (22,801)	376.89	0
201 x 201 (40,401)	672.01	0
251 x 251 (63,001)	1,027.27	0

Table 4. IO Monad experiments on varying grid sizes on 4 cores.

Grid-Size	μ Duration (sec)	σ Duration (sec)
51 x 51 (2,601)	38.614	0.397
101 x 101 (1,0201)	171.61	3.016
151 x 151 (22,801)	404.11	0
201 x 201 (40,401)	720.65	0
251 x 251 (63,001)	1,117.27	0

Table 5. IO Monad experiments on varying grid sizes on 3 cores.

Grid-Size	μ Duration (sec)	σ Duration (sec)
51 x 51 (2,601)	**10.822**	0.377
101 x 101 (10,201)	107.40	1.306
151 x 151 (22,801)	464.017	0
201 x 201 (40,401)	1,227.68	0
251 x 251 (63,001)	3,283.63	0

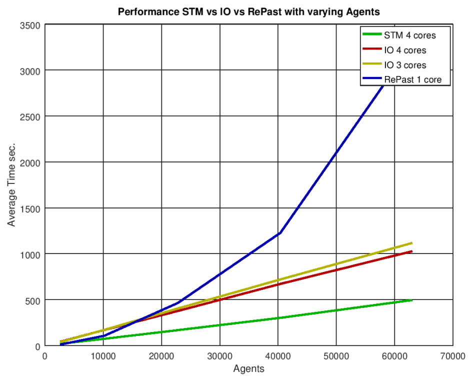
Table 6. Repast experiments on varying grid sizes on a single (1) core.

(1) core Java RePast version outperforms the 4 core Haskell STM version by about 200%. The figure is inverted on a 251x251 grid where the 4 core Haskell STM version outperforms the single core Java Repast version by over 600%. 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 over 600%.

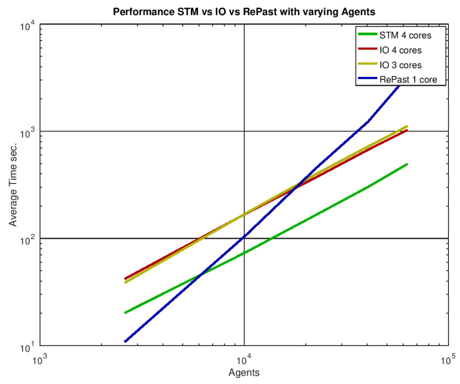
5.4 Retries

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 7.

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 IO 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.



(a) Normal Scale



(b) Logarithmic scale on both axes

Fig. 3. Comparison of STM (Table 3), IO (Table 4, Table 5) and RePast (single core) (Table 6) performance. TODO: re-create the figure when all experiments had 8 runs.

Grid-Size	Commits	Retries Avg (Std)	Ratio
51 x 51 (2,601)	2,601,000	1306.5 (83.9)	0.0
101 x 101 (10,201)	10,201,000	3712.5 (308.42)	0.0
151 x 151 (22,801)	22,801,000	8189.5 (342.12)	0.0
201 x 201 (40,401)	40,401,000	13285 (0.0)	0.0
251 x 251 (63,001)	63,001,000	21217 (0.0)	0.0

Table 7. Retries Ratio of STM Monad experiments on varying grid sizes on 4 cores.

5.5 Insights

Interpretation 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 IO 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 (SECOND ENCOUNTER)

One of the first Agent-Based Simulation model which rose to some prominence was the Sugarscape model developed by Epstein and Axtell in 1996 [8]. Their aim was to *grow* an artificial society by simulation and connect observations in their simulation to phenomenon of real-world societies. The main features of this model are:

- Searching, harvesting and consuming of resources.
- Wealth and age distributions.
- Seasons in the environment and migration of agents.
- Pollution of the environment.
- Population dynamics under sexual reproduction.
- Cultural processes and transmission.
- Combat and assimilation.
- Bilateral decentralized trading (bartering) between agents with endogenous demand and supply.
- Emergent Credit-Networks.
- Disease Processes, Transmission and immunology.

Because of its essential importance to this field, its complexity, number of features and allowing us to bridge the gap to ACE, we select it as the first of two central models, which will serve as use-case to develop our methods. The idea is to formally specify and then verify the process of bilateral decentralized trading because it is the most complex of the features and connects directly to ACE.

We implemented Chapter II of the book. TODO: shortly explain how agents behave

The model specification requires to shuffle agents before every step. This happens automatically due to race-conditions in concurrency we arrive at an effectively shuffled processing of agents: we can assume that the order of the agents is *effectively* random in every step - with the important difference, that we do not have control over this randomness as we would have when shuffling.

Note that in contrast to the SIR case-study we don't provide an IO implementation because we focus on different thing here. The focus here is on how different data-structures can make a huge impact on the performance.

TODO: show a picture generated by our software after initialisation, and then after the number of agents has stabelised

6.1 Experiment Design

We follow [13] and measure the average updates per second of the simulation with a 50x50 environment and an initial population of 500 over 60 seconds. We parametrise the model with the following configuration from chapter II of the book, of the section *Carrying Capacity* (p. 30): movement, harvest, regrow, not dying of age, no seasons, no pollution, no inheritance. give the exact parameters. In this case the authors of the Sugarscape book report that the initial number of agents quickly drops and stabilises, which is in unison with our results as we show in Figure TODO. This behaviour also guarantees that we don't run out of agents and it shows the highly dynamic nature of the model and the ability of our implementation to quickly spawn and terminate threads.

For each experiment we conducted 8 runs on our machine (see Table 1) under no additional work-load and report both the average and standard deviation. 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 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. TODO main measure: steps/sec and retry-ratio

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 what we are interested here thus we completely omit it and only output

	Cores	Steps	Ratio
State	1	1,693.1 (8.305)	28.219 (0.138)
STM	1	1,854.2 (29.519)	30.904 (0.491)
STM	2	2,129.5 (61.414)	35.492 (1.023)
STM	3	2,312.5 (71.658)	38.542 (1.194)
STM	4	2,238.8 (36.307)	37.312 (0.605)

Table 8. Performance on 50x50 grid and 500 initial agents with varying number of cores.

Cores	Commits	Retries	Ratio
1	498,881 (3,042.4)	2,407.8 (248.56)	0.004
2	557,800 (7,503.3)	592,890 (8,827.9)	1.062
3	540,700 (7,530.2)	1,189,100 (19,771)	2.199
4	486,850 (7,927.9)	1,646,800 (29,939)	3.382

Table 9. Retries on 50x50 grid and 500 initial agents with varying number of cores.

Cores	Steps	Ratio
1	1,868.8 (19.129)	31.146 (0.318)
2	2,121.1 (36.294)	35.352 (0.604)
3	2,325 (53.570)	38.750 (0.892)
4	2,245 (40.175)	37.417 (0.669)

Table 10. Performance on 50x50 grid and 500 initial agents with varying number of cores with a synchronous environment.

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 Naive Approach using TVar and concurrent Environment

Experiments with varying number of cores. The results are reported in Table **??**.

state: shuffles agents after every step environment in STM is run as concurrent agent

We also compared the average retry-ratio on varying number of cores.

6.3 Running Environment non-concurrently

Running concurrently is strictly speaking wrong because could lead to runs where the regrowth happens after the agent harvests which violates the model specifications.

We also compared the average retry-ratio on varying number of cores.

mean (commits) = 500230 std (commits) = 22588

mean (retries) = 1065700 std (retries) = 67492

mean (retries) / mean (commits) = 2.1304

Seems to have no effect, can omit it and replace the results when running with environment

Cores	Commits	Retries	Ratio
1	482,790 (23,061)	2,397.9 (263.88)	0.004
2	500,300 (20,063)	526,000 (21,586)	1.0514
3	500,230 (22,588)	1,065,700 (67,492)	2.1304
4			

Table 11. Retries on 50x50 grid and 500 initial agents with varying number of cores with a synchronous environment.

Cores	Steps	Ratio
1	3,047.8 (11.042)	50.796 (0.184)
2	4,127.2 (31.824)	68.788 (0.530)
3	4,641.2 (31.459)	77.354 (0.524)
4	5,215.5 (29.057)	86.925 (0.484)

Table 12. Performance on 50x50 grid and 500 initial agents with varying number of cores using a *TArray* for a synchronous environment.

Cores	Commits	Retries	Ratio
1			
2			
3			
4			

Table 13. Retries on 50x50 grid and 500 initial agents with varying number of cores using a *TArray* for a synchronous environment.

Cores	Steps	Ratio
1	3,342.4 (32.350)	55.706 (0.539)
2	3,168.5 (60.387)	52.808 (1.006)
3	3,861.4 (135.97)	64.356 (2.266)
4	3,981.1 (65.951)	66.352 (1.0992)

Table 14. Performance on 50x50 grid and 500 initial agents with varying number of cores using a *TArray* for a concurrent environment.

6.4 From TVar to TArray

Sync Environment: runs after all agents.

We also compared the average retry-ratio on varying number of cores.

————————————— Concurrent Environment

We also compared the average retry-ratio on varying number of cores.

6.5 Scaling up Agents

So far we always kept the initial number of agents at 500 and due to the model specification the number drops quickly to around 250-270 and stabelises around there due to the carrying capacity of the environment as described in the Book on page TODO. We now want to see the scaling

Cores	Commits	Retries	Ratio
1			
2			
3			
4			

Table 15. Retries on 50x50 grid and 500 initial agents with varying number of cores using a *TArray* for a concurrent environment.

Agents	Steps	Ratio
500	847.25 (5.775)	14.121 (0.096)
1,000	410.5 (5.529)	6.841 (0.092)
1,500	270.0 (6.989)	4.5 (0.116)
2,000	198.12 (4.223)	3.302 (0.07)
2,500	158.50 (7.764)	2.641 (0.129)

Table 16. Performance on 50x50 grid and varying number of agents using the State approach.

Agents	Steps	Ratio
500	1,270.9 (14.337)	21.181 (0.238)
1,000	679.50 (18.142)	11.325 (0.302)
1,500	490.62 (2.133)	8.177 (0.035)
2,000	377.12 (0.640)	6.285 (0.010)
2,500	316.38 (1.060)	5.272 (0.017)

Table 17. Performance on 50x50 grid and varying number of agents using the TVar approach.

Agents	Steps	Ratio
500	4,468.2 (67.753)	74.471 (1.129)
1,000	3,410.5 (16.449)	56.842 (0.274)
1,500	2,717.9 (12.955)	45.298 (0.215)
2,000	2,221.2 (13.446)	37.021 (0.224)
2,500	1,904 (19.346)	31.733 (0.322)

Table 18. Performance on 50x50 grid and varying number of agents using the TArray approach.

property of our approaches when increasing the number of agents. For this we slightly change the implementation: always when an agent dies it spawns a new one. This ensures that we keep the number of agents always constant (fluctuactes slightly between 500 and 497) over the whole duration. This ensures a constant load of concurrent processes interacting with each other and demonstrates also the ability to terminate and fork threads dynamically during the simulation. Except for the State approach we run all experiments with 4 cores. Also in this case the environment is run after all agents are run. We look into the performance of 500, 1,000, 1,500, 2,000 and 2,500 (maximum possible capacity of the 50x50 environment).

6.6 Insights

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 indeed, the performance goes down dramatically as reported in table TODO when we increase the environment to 128x128 with same number of agents. Obviously this is the result of Amdahls law where the environment becomes the limiting factor of the simulation. Depending on the underlying data-structure used for the environment we have to options. In the case of the State and TVar implementation we build on an indexed array which we can updated in parallel using the existing data-parallel support in Haskell (TODO: explain). 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.

Comparison with imperative approaches: they are running it on 128x128 and on 10 year old single-core machines [13]

Interpretation of the performance data leads to the following insights:

7 CONCLUSION (THE MORAL OF THE TALE)

Using STM for concurrent, large-scale ABS seems to be a very promising approach as our proof-of-concept has shown. The concurrency abstractions of STM are very powerful, yet simple enough to allow convenient implementation of concurrent agents without the nastiness of low level concurrent locks. Also we have shown by experiments, that we indeed get a very substantial speed-up and that we even got linear performance scaling for our model.

Interestingly, STM primitives map nicely to ABS concepts: using a share environment through a *TVar* is very easy, also we implemented in an additional proof-of-concept the use of *TChan* 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.

Running in STM instead of IO also 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 Array within a TVar, one can reduce retries significantly. We tracked the retries in our proof-of-concept using the stm-stats library and arrived at a ratio of 0.0% retries - note that there were some retries but they were so low that they weren't significant.

Benefits are that using STM takes a big portion of burden from the modeller as one can think in STM primitives instead of low level locking and concurrency operational details.

8 FURTHER RESEARCH (LIVED HAPPILY EVER AFTER...)

Despite the promising proof-of-concept, still there is more work needed:

- implement other Sugarscape chapers: future research, also need look at more more models
- We have not focused on implementing an approach like *Sense-Think-Act* cycle as mentioned in [21]. This could offer lot of potential for parallelisation due to sense and think happening

isolated for each agent without interfering with global shared data. We expect additional speed-up from such an approach but leave this for further research.

- So far we only looked at a time-driven model. It would be of fundamental interest whether we can somehow apply STM and concurrency to an event-driven approach as well. We hypothesise that it is not as striking and easy due to the fundamental sequential approach to even-processing. Generally one could run agents concurrently and undo actions when there are inconsistencies - something which STM supports out of the box. atm it is a time-driven lock-step approach. it would be interesting to see how an event-driven approach through an underlying PDES implementation would perform
- So far we only looked at asynchronous agent-interactions through TVar and TChan: agents modify the data or send a message but don't synchronise on a reply. Also a receiving agent doesn't do synchronised waiting for messages or data-changes. Still, in some models we need this synchronous way of agent-interactions where agents interact over multiple steps within the same global time-step. We yet have to come up with an easy-to-use solution for this problem using STM.
- Partitioning the environment into subsets which can be updated concurrently / parallel could speed up the environment updating as well. Is particularly easy in FP and using STM TArray.
- going towards distribution using Cloud haskell.
- Amazon AWS allows to scale up to potentially thousands of cores - it would be highly interesting to see the performance of STM there. Also it would be of interest to see how well it scales to thousands of cores and investigate where the limit is when performance begins to decrease due to increasing numbers of retries.

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] Joe Armstrong. 2010. Erlang. *Commun. ACM* 53, 9 (Sept. 2010), 68–75. https://doi.org/10.1145/1810891.1810910
[3] Nikolaos Bezirgiannis. 2013. *Improving Performance of Simulation Software Using Haskells Concurrency & Parallelism.* Ph.D. Dissertation. Utrecht University - Dept. of Information and Computing Sciences.
[4] Fernando Castor, JoÃčo Paulo Oliveira, and AndrÃĺ L.M. Santos. 2011. Software Transactional Memory vs. Locking in a Functional Language: A Controlled Experiment. In *Proceedings of the Compilation of the Co-located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11 (SPLASH '11 Workshops).* ACM, New York, NY, USA, 117–122. https://doi.org/10.1145/2095050.2095071
[5] 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
[6] 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
[7] Richard H. Enns. 2010. *It's a Nonlinear World* (1st ed.). Springer Publishing Company, Incorporated.
[8] Joshua M. Epstein and Robert Axtell. 1996. *Growing Artificial Societies: Social Science from the Bottom Up.* The Brookings Institution, Washington, DC, USA.
[9] 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
[10] Tim Harris and Simon Peyton Jones. 2006. Transactional memory with data invariants. https://www.microsoft.com/en-us/research/publication/transactional-memory-data-invariants/
[11] 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

[12] 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

[13] 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

[14] 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

[15] 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

[16] Cristian Perfumo, Nehir SÃűnmez, Srdjan Stipic, Osman Unsal, AdriÃąn 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

[17] Alban Rousset, BÃĺnÃľdicte Herrmann, Christophe Lang, and Laurent Philippe. 2016. A survey on parallel and distributed multi-agent systems for high performance computing simulations. *Computer Science Review* 22 (Nov. 2016), 27–46. https://doi.org/10.1016/j.cosrev.2016.08.001

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

[19] 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

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

[21] 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