Chapter 6

Parallel ABS

The future of scientific computing in general and Agent-Based Simulation (ABS) in particular is parallelism: Moore's law is declining as we are reaching the physical limits of CPU clocks. The only option is to go massively parallel due to the availability of cheap parallel local hardware with many cores, or cloud services like Amazon EC. This trend has already been recognised in the field of ABS as a research challenge for Large-scale ABMS [117] was called out and as a substantial body of research for parallel ABS shows [1, 31, 60, 67, 76, 81, 108, 114, 129, 153, 169, 170].

In this body of work, it has been established that parallelisation of autonomous agents, situated in some spacial, metric environment can be particularly challenging. The reason for this is that the environment constitutes a key medium for the agents' interactions, represented as a passive data structure, recording attributes of the environment and the agents [108]. Thus, the problem of parallelising ABS boils down to the problem of how to synchronise access to shared state, without violating the causality principle and resource constraints [114, 170]. Various researchers have developed different techniques and most of them are based on the concept of Parallel Discrete-Event Simulation (PDES). The idea behind PDES is to partition the shared space into logical processes, which run at their own speed, processing events coming from themselves and other logical processes. To deal with inconsistencies a conservative approach exists, which does not allow for processing of events with a lower timestamp than the current time of the logical process. It is an optimistic approach, which deals with inconsistencies through rolling back changes to state.

Adopting PDES to ABS is challenging, as agents are autonomous, which means that the topology can change in every step. This erractic set of changes makes it hard to predict the topology of logical processes in advance [108], posing a difficult problem for parallelisation in general [31]. The work [169, 170] discusses this challenge by giving a detailed and in-depth discussion of the internals and implementation of their powerful and highly complex PDES-MAS system. The rather conceptual work [114] proposes a general, distributed simulation framework for multi-agent systems and addresses a number of key problems: decomposition of the environment, load balancing, modelling, communication and shared state variables, which the authors mention as the central problem of parallelisation.

In addition, various distributed simulation environments for ABS have been developed and their internals published in research papers: the SPADES system [153] manages agents through UNIX pipes using a parallel sense-think-act cycle and employing a conservative PDES approach. Mace3J [60] is a Java based system running on single or multi-core workstations. It implements a message passing approach to parallelism. James II [81] is also a Java based system and focuses on a PDEVS simulation with a plugin architecture to facilitate the reuse of models. The well known RePast-HPC [67, 129] framework uses a PDES engine under the hood.

The baseline of this body of research is that parallelisation is possible and we know how to do it. However, the complexity of these parallel and distributed simulation concepts and toolkits is high and the model development effort is difficult [1]. Furthermore, this

Formatted: English (UK)

Formatted: English (UK)

Deleted: been already

Commented [KJ1]: I don't understand what you are trying to say here. "recognised in the field of ABS as a large-scale research challenge and was called out, as a substantial body of research…shows?" Right now it doesn't make sense.

Deleted: , where

Deleted: there exists

Deleted: to process

Deleted: ;

Deleted: and

Deleted: ,

Deleted: making

Deleted: ;

Deleted: -

Deleted: ;

Deleted: ;

Deleted: t

Deleted: is using

Deleted: hard

sophisticated and powerful machinery is not always required as ABS does not always need to be run in a distributed way. However, the implementers 'simply' want to parallelise their models locally. Although these existing distributed ABS frameworks could be used for to parallelise local models, they are an excess and more straightforward concepts for parallelising ABS would be appropriate. That being said, in this case not very much research exists, and implementers either resort to the distributed ABS frameworks, implement their own low-level, complex concurrency plumbing, or simply refrain from using parallelism due to the complexity and accept a longer execution time. What makes it worse is that parallelism always comes with the danger of very subtle bugs, which might lie dormant, potentially invalidating significant scientific results of the model. Therefore, something simpler is needed for local parallelism. Unfortunately, the established imperative languages in the ABS field, Python, Java, C++, don't make adding parallelism easy, due to their inherent use of unrestricted side effects. What is more, they mostly follow a lock-based approach to concurrency which is error prone and does not compose.

Deleted: but

Deleted: this

Deleted: However

Deleted: for

Deleted: there does not exist

Deleted: much research

Deleted: which can be considerably complex -

Deleted: high

Deleted: involved

Deleted: additional,

Deleted: Further

Chapter 8

Concurrency in ABS

In an ideal world, we would like to solve all our problems using parallelism but unfortunately, it can't be applied to all parallel problems and ABS is no exception. As soon as there are data dependencies we cannot avoid concurrency. This problem can be seen in the Sugarscape model, in the form of the mutable environment and synchronous agent interactions, and to a lesser extent in the monadic SIR with the Rand Monad. More generally, this issue is due to the fact that agents are executed within a monadic context, from which the sequencing of effective computations immediately follow. This is the very meaning of the Monad abstraction. Indeed, we have shown, both by argument and measurement in the previous chapter, the very fact that parallelism is simply not applicable to monadic execution of agents due to sequencing of effects. This fact renders all attempts of running monadic agents in parallel void. In this chapter we discuss the use of concurrency to run agents with a monadic context in parallel, which is the only way we can execute monadic agents at the same time.

Deleted: ,

Deleted: like we have them

Deleted: , we cannot avoid concurrency

Deleted: s

Deleted: ful

Deleted: follows

Deleted: ,

Deleted: ,

Deleted: which

Traditional approaches to concurrency follow a lock-based approach, where sections which access shared data are synchronised through synchronisation primitives like mutexes, semaphores or monitors. However, with the established programming languages in the field, Python, Java and C++, it is not easy to address the complexities of parallel programming due to unrestricted side effects and the intricacies of low-level locking semantics. Therefore, in this chapter we follow a different path and look into using Software Transactional Memory (STM) for implementing concurrent ABS, which promises to overcome the problems of lock-based approaches. We hypothesise that by using STM in Haskell, implementing local parallel ABS is considerably easier than with lock-based approaches, less error prone, and easier to validate.

Our hypothesis is supported by [45], which gives a good indication as to how difficult and complex constructing a correct concurrent program can be. In addition, [45] shows how much easier, concise and less error prone an STM implementation is over traditional locking with mutexes and semaphores. More importantly, it indicates that STM consistently outperforms the lock-based implementations. We follow

Deleted: is

Deleted: and

Deleted: shows

this work and compare the performance of lock-based and STM implementations and hypothesise that the reduced complexity and increased performance will be directly applicable to ABS as well.

The idea of applying transactional memory to simulation was already explored in the work [76], where the authors looked into how to apply Intel's hardware transactional memory to simulations in the context of a Time Warp Parallel Discrete Event Simulation (PDES). The results showed that their approach generally outperformed traditional locking mechanisms. Although PDES is a different problem than ABS, they are related as both deal with events (in case of an event-driven ABS) and PDES has been successfully adopted to simulate parallel ABS [170].

To test our hypothesis, we present two case studies using the already introduced SIR (Chapter 2.1.1) and Sugarscape (Chapter 2.1.2) models. We compare the performance of lock-based and STM implementations in each case where we investigate both the scaling performance under increasing number of CPU cores and agents. We show that the STM implementations consistently outperform the lock-based ones and scale much better to increasing number of CPU cores both on local hardware and on Amazon EC services.

Unfortunately, as soon as we employ concurrency, we lose all static guarantees about reproducibility and the use of STM is no exception. Still, STM has the unique benefit that it can guarantee a lack of persistent side effects at compile time, allowing unproblematic retries of transactions, something of fundamental importance in STM as will be described below. This also implies another very compelling advantage of STM over unrestricted lock-based approaches. By using STM, we can reduce the side effects allowed substantially and guarantee at compile time that the differences between runs in the same initial conditions will only stem from the fact that we run the simulation concurrently, and from nothing else. All this makes the use of STM very compelling and to our best knowledge we are the very first to investigate the use of STM for implementing concurrent ABS in a systematic way.

Deleted: the

Deleted: implies also

Deleted: ,

Deleted: of

8.1 Software Transactional Memory

Software Transactional Memory was introduced by [161] in 1995 as an alternative to lock-based synchronisation in concurrent programming which, in general, is notoriously difficult to get right. This is because reasoning about the interactions of multiple concurrently running threads and low-level operational details of synchronisation primitives is very hard. The main problems are:

Deleted:

- Race conditions due to forgotten locks;
- Deadlocks resulting from inconsistent lock ordering;

- Corruption caused by uncaught exceptions;
- Lost wake-ups induced by omitted notifications.

Formatted: Font: CMR10

What is worse, concurrency does not compose. It is very difficult to write two functions (or methods in an object) acting on concurrent data which can be composed into a larger concurrent behaviour. The reason for the difficulty is that one has to know about the internal details of locking, which breaks encapsulation and makes composition dependent on knowledge about their implementation. Therefore, it is impossible to

Deleted: Worse

Deleted: it

compose two functions where, for example, one withdraws some amount of money from an account and the other deposits this amount of money into a different account. The problem is that one ends up with a temporary state where the money is in neither of the accounts, creating an inconsistency and a potential source for errors because threads can be rescheduled at any time.

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

8.1.1 Software Transactional Memory in Haskell

The work of [73, 74] added STM to Haskell, which was one of the first programming languages to incorporate STM with composable operations into its main core. Various implementations of STM exist in other languages as well (Python, Java, C#, C/C++). But, we argue that it is in Haskell with its type system and the way in which side effects are treated is where it truly shines.

In the Haskell implementation, STM actions run within the STM Monad. This restricts the operations to only STM primitives as shown below. This means that STM actions are always repeatable without persistent side effects because such persistent side effects (for example writing to a file, launching a missile) are not possible in the STM Monad. This is also the fundamental difference to IO, where all bets are off and everything is possible because IO can run everything without restrictions.

Thus, the ability to restart an action without any persistent effects is only possible due to the nature of Haskell's type system and by restricting the effects to STM only, ensures that only controlled effects, which can 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;

• TMVar - a transactional synchronising variable which is either empty or full. To read from an empty or write to a full, TMVar will cause the current thread to block and retry its transaction when any transactional primitive of this action has changed.

- TArray - a transactional array where each cell is an individual transactional variable TVar, allowing more finer-grained transactions instead of having the whole array in a TVar.
- TChan - a transactional channel, representing an unbounded FIFO channel, based on a linked list of TVar.

Deleted: none of either

Deleted: ,

Deleted:

Deleted: as well

Deleted: -

Deleted: There exist v

Deleted: b

Deleted: how

Deleted: ,

Deleted: which allows to enforce that

Deleted: because

Deleted: as there are basically no restrictions

Deleted: much

Furthermore, STM also provides combinators to deal with blocking and composition:

- retry :: STM () retries an STM action. This will cause the program to abort the current transaction and block the thread where it is running. When any of the transactional data primitives have changed, the action will be run again. This is useful to await the arrival of data in a TVar, or put more generally, to block arbitrary conditions.
- orElse :: STM a → STM a → STM a allows us to combine two blocking actions where either one is executed, but not both. The first action is run and if it is successful its result is returned. If it retries, then the second is run and if that one is successful its result is returned. If the second one retries, the whole orElse retries. This can be used to implement alternatives in blocking conditions, which can obviously be nested arbitrarily.

To run an STM action the function atomically :: STM a → IO a is provided, which performs a series of STM actions atomically within the IO Monad. It takes the STM action, which returns a value of type a and returns an IO action which returns a value of type a. The IO action then can only be executed from within the IO Monad, either within the main thread or an explicitly forked thread.

STM in Haskell is implemented using optimistic synchronisation, which means that instead of locking access to shared data, each thread keeps a transaction log for each read and write to shared data that it makes. When the transaction exits, the thread checks whether it has a consistent view to the shared data or not. It checks whether other threads have written to memory it has read, thus it can identify whether a rollback is required or not.

However, STM does not come without issues. The authors of [148] analyse several Haskell STM programs with respect to their transactional behaviour. They identified the roll-back rate as one of the key metrics, 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 is high.

8.1.2 STM Examples

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

```
transferFunds :: TVar Integer -> TVar Integer -> Integer -> IO ()
```

Deleted: in
Deleted: s
Deleted: on
Deleted: to combine
Deleted: s
Deleted: appears to be
Deleted: e
Formatted: Font: CMTT10

```
transferFunds from to n = atomically (do
  withdraw from n
  deposit to n)
withdraw :: TVar Integer -> Integer -> STM ()
withdraw account amount = do
  balance <- readTVar account
  writeTVar (balance - amount)
deposit :: TVar Integer -> Integer -> STM ()
deposit account amount = do
  balance <- readTVar account
  writeTVar (balance + amount)
```

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

Deleted: ,

```
stmAction :: TVar Int -> StateT Int STM Int
stmAction v = do
  -- print a debug output and increment the value in StateT

Debug.trace "increment!" (modify (+1))
-- read from the TVar
n <- lift (readTVar v)
-- await a condition: content of the TVar >= 42 if n < 42

   -- condition not met: retry

then lift retry
-- condition met: return content ot TVar else return n
```

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

Deleted: ¶

Deleted: f

```
stmThread :: TVar Int -> IO ()
stmThread v = do
  -- the initial state of the StateT transformer

let s = 0
-- run the state transformer with initial value of s (0) let ret = runStateT (stmAction v) s
```

```
-- atomically run the STM block
(a, s') <- atomically ret
-- print final result
putStrLn("final StateT state = " ++ show s' ++

        ", STM computation result = " ++ show a)
```

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

Deleted: d

```
main :: IO ()
main = do
  -- create a new TVar with initial value of 0

v <- newTVarIO 0
-- start the stmThread and pass the TVar forkIO (stmThread v)
-- do 42 times...
forM_ [1..42] (\i -> do

    -- use delay to 'make sure' that a retry is happening for ever increment

threadDelay 10000
-- write new value to TVar using atomically atomically (writeTVar v i))
```

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

Deleted: -

8.2 Software Transactional Memory in ABS

In this section we give a short overview of how we apply STM to pure functional ABS. In both case studies we fundamentally follow a time-driven, parallel ap-

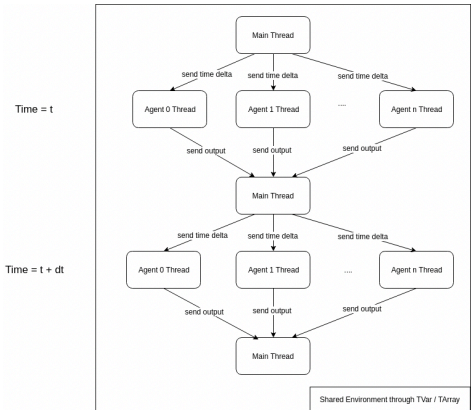
Figure 8.1: Diagram of the parallel time-driven lock-step approach.

proach as introduced in Chapter 2.3.2, where the simulation is advanced by a 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 8.1 for a visualisation of the concurrent, parallel time-driven lock-step approach.

By running each agent in a thread will guarantee the execution in parallel even if the agent has a monadic context. This forces us to evaluate each agent's monadic context separately instead of running them all in a common context. This means that we are ending up in the IO Monad, because STM can be only transacted from within an IO context, due to non-deterministic side effects. This is no contradiction to our original claim. Yes, we are running in IO but the agent behaviour itself is not, which is a fundamental difference.

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

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



8.2.1 Adding STM to Agents

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

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

Deleted: not the

Deleted: ,

Deleted: f

Deleted: ,

Deleted: through which

Deleted: t

Deleted: a

Deleted: -

Deleted: a

Deleted: the

Deleted: Further

```
-- Make Rand a transformer to be able to add STM as innermost monad

type SIRMonad g = RandT g STM
-- Input to agent is now an empty tuple instead of the Environment type SIRAgent g = SF
(SIRMonad g) () SIRState

-- The MSF construction function takes now the TVar with the environment.
sirAgent :: RandomGen g => TVar SIREnv -> Disc2dCoord -> SIRState -> SIRAgent g

-- The infected agent behaviour is nearly the same except that
-- the agent modifies the environment through the TVar
infected :: RandomGen g => SF (SIRMonad g) () (SIRState, Event ()) infected = proc _ ->
do

  recovered <- occasionally illnessDuration () -< ()
  if isEvent recovered

then (do
-- update the environment through the TVar
arrM_ (lift $ lift $ modifyTVar env (changeCell coord Recovered)) -< () returnA -<
(Recovered, Event ()))

    else returnA -< (Infected, NoEvent)
```

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

```
agentThread :: RandomGen g
          => Int

-> SIRAgent g
->g
-> MVar SIRState -> MVar DTime
-> IO ()
0 _ _ _ _ = return

-- ^ Number of steps to compute
-- ^ Agent behaviour MSF
-- ^ Random-number generator of the agent
-- ^ Synchronisation back to main thread
-- ^ Receiving DTime for next step
agentThread
agentThread n sf rng retVar dtVar = do

() -- all steps computed, terminate thread -- wait for dt to compute current step

dt <- takeMVar dtVar
-- compute output of current step
let sfReader = unMSF sf ()
```

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



```
 -- post result to main thread
 putMVar retVar ret
 -- tail recursion to next step
 agentThread (n - 1) sf' rng' retVar dtVar
```

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

```
simulationStep :: TVar SIREnv -- ^ environment
-> [MVar DTime] -- ^ sync dt to threads

-> [MVar SIRState] -- ^ sync output from threads -> DTime -- ^ time delta
-> IO SIREnv

simulationStep env dtVars retVars dt = do
-- tell all threads to continue with the corresponding DTime mapM_ (`putMVar` dt)
dtVars
-- wait for results but ignore them, SIREnv contains all states mapM_ takeMVar retVars
-- return state of environment when step has finished readTVarIO env
```

The difference to an implementation which uses IO are minor but far reaching. Instead of using STM as innermost Monad, we use IO, thus running the whole agent behaviour within the IO Monad. Instead of receiving the environment through a TVar, the agent receives it through an IORef. It also receives an MVar, which is the synchronisation primitive to synchronise the access to the environment in the IORef amongst all agents. Agents grab and release the synchronisation lock of the MVar when they enter and leave a critical section in which they operate on the environment stored in the IORef.

8.3 Case Study I: SIR

Deleted: s

Our first case study is the SIR model as introduced in Chapter 2.1.1. The aim of this case study is to investigate the potential speedup a concurrent STM implementation gains over a sequential one under a varying number of CPU cores and agent populations. The behaviour of the agents is quite simple and the interactions are happening indirectly through the environment, where reads from the environment outnumber the writes to it by far. Furthermore, a comparison to lock-based implementations with the IO Monad is done to show that STM outperforms traditional lock-based concurrency in a functional ABS implementation while still retaining some static guarantees.

8.3.1 Experiment Design

In this case study we compared the performance of five (5) implementations under varying numbers of CPU cores and agent numbers. The code of all implementations can be accessed freely from the code repository [175].

1. Sequential - This is the reference implementation as discussed in Chapter 4.3, where the agents are executed sequentially within the main thread without any concurrency. The discrete 2D grid is represented using an indexed array [109] and shared amongst all agents as read-only data, with the main thread updating the array for the next time step.
2. Lock-Based Naive - This is the same implementation as Sequential, but the agents now run concurrently in the IO Monad. The discrete 2D grid is also represented using an indexed array, but is now modified by the agents themselves and therefore shared using a global reference. The agents acquire and release a lock when accessing the shared environment.
3. Lock-Based Read-Write Lock - This is the same implementation as Lock-Based Naive, but uses a read-write lock from concurrent-extra library [188] for a more fine-grained locking strategy. This implementation exploits the fact that in the SIR model, reads outnumber writes by far, making a read-write lock much more appropriate than a naive locking mechanism, which unconditionally acquires and releases the lock. However, it is important to note that this approach works only because the semantics of the model support it: agents read any cells but only write their own cell.
4. Atomic IO - This is the same implementation as Lock-Based Read-Write Lock but uses an atomic modification operation to both read and write the shared environment. Although it runs in the IO Monad, it is not a lock-based approach as it does not acquire locks. Instead it uses a compare-and-swap hardware instruction. A limitation of this approach is that it is only applicable when there is just a single reference in the program and that all operations need to go through the atomic modification operation. As in the case of the Lock-Based Read-Write Lock implementation, this approach works only because the semantics of the model support it.
5. STM - This is the same implementation as Lock-Based Naive but agents run in the STM Monad. The discrete 2D grid is also represented using an indexed array, but it is shared amongst all agents through a transactional variable TVar.

Each experiment was run on our hardware (see Table 8.1) under no additional workload until t = 100 and stepped using Δt = 0.1. In the experiments we varied the number of agents (grid size) as well as the number of cores when running concurrently. We checked the visual outputs and the dynamics and they look qualitatively the same as the reference Sequential implementation [179]. A rigorous, statistical comparison of all implementations, to investigate the effects

Deleted:

Deleted:

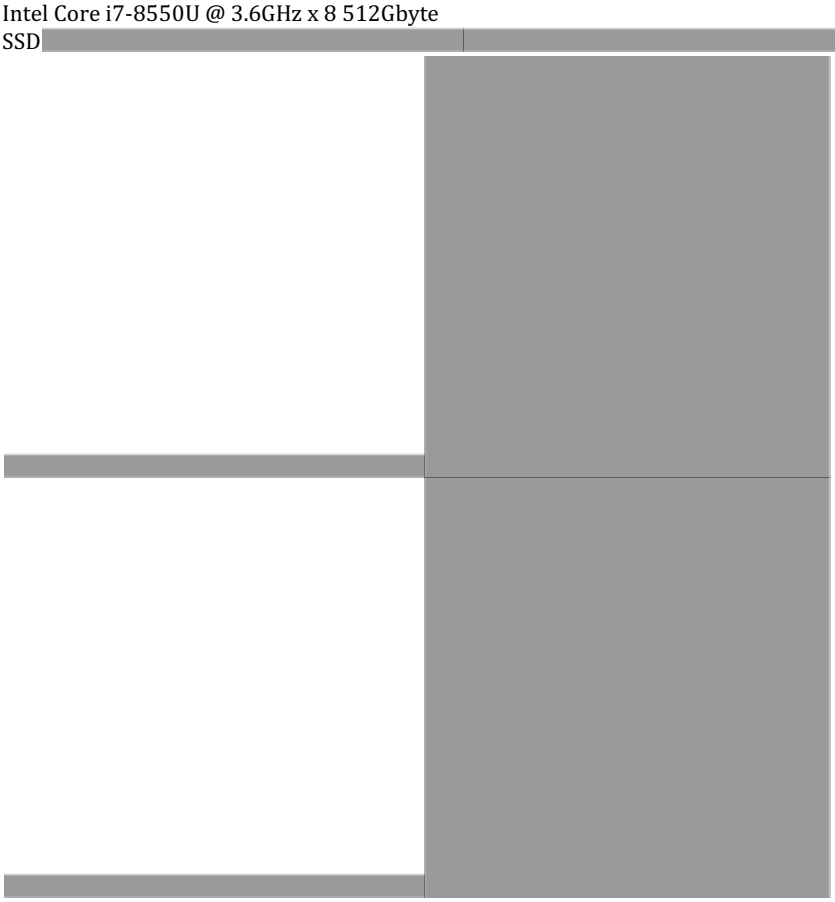
Deleted: but

Deleted:

Model OS RAM CPU HD

Dell XPS 13 (9370) Ubuntu 19.10 64-bit
16 GByte

Intel Core i7-8550U @ 3.6GHz x 8 512Gbyte
SSD



Haskell
GHC 8.4.3 (stack resolver lts-12.4)

Table 8.1: Hardware and software details for all experiments

Cores Sequential Lock-Based Naive Lock-Based Read-Write 1

2 3 4 5 6 7 8

Atomic IO

STM 52.2 (0.23) 33.2 (0.03) 26.4 (0.05) 23.3 (0.19) 23.0 (0.06) 23.1 (0.05) 23.4 (0.22) 26.2 (0.22)

73.9 (2.06)	59.2 (0.16)	55.0 (0.22)	51.0 (0.11)
-	46.5 (0.05)	40.8 (0.18)	32.4 (0.09)
-	44.2 (0.08)	35.8 (0.06)	25.5 (0.09)

-	47.4 (0.12)	34.0 (0.32)	22.7 (0.08)
-	48.1 (0.13)	34.5 (0.06)	22.6 (0.03)
-	49.1 (0.09)	34.8 (0.03)	22.3 (0.09)
-	49.8 (0.09)	35.9 (0.15)	22.8 (0.07)
-	57.2 (0.06)	40.4 (0.21)	25.8 (0.02)

Table 8.2: Performance comparison of Sequential, Lock-Based, Atomic IO and STM SIR implementations under varying cores with grid size of 51x51 (2,601) agents. Timings in seconds (lower is better), standard deviation in parentheses.

of concurrency on the dynamics, is quite involved and therefore beyond the focus of this thesis. But as a remedy, we refer to the use of property-based testing, as shown in Chapter 11.

Deleted: b

For robust performance measurements we used the microbenchmarking library Criterion [138, 139]. It allows the definition and running of benchmark suites, measuring performance by executing them repeatedly, fitting actual against expected runtime, reporting mean and standard deviation for statistically robust results. By running each benchmark repeatedly, and fitting it using linear regression analysis, Criterion is able to robustly determine whether the measurements fall within a normal range or are outliers (and therefore should be re-run) due to some external influences like additional workload on the machine. Therefore, we made sure only to include measurements Criterion labelled as 'normal', which meant we re-ran measurements where goodness-of-fit was $R^2 < 0.99$. Criterion ran each of our benchmarks 10 times with increasing increments of 1, 2, 3 and 4 times. In the results we report the estimates of ordinary least squares (OLS) regression together with the standard deviation because it gives the most reliable results in terms of statistical robustness.

Deleted: to

8.3.2 Constant Grid Size, Varying Cores

Deleted: g
Deleted: s
Deleted: v
Deleted: c

In this experiment we held the grid size constant at 51 x 51 (2,601 agents) and varied the cores where possible. The results are reported in Table 8.2 and visualised in Figure 8.2.

Figure 8.2: Performance comparison of Sequential, STM, Lock-Based and Atomic IO SIR implementations on varying cores with grid size of 51x51 (2,601) agents.

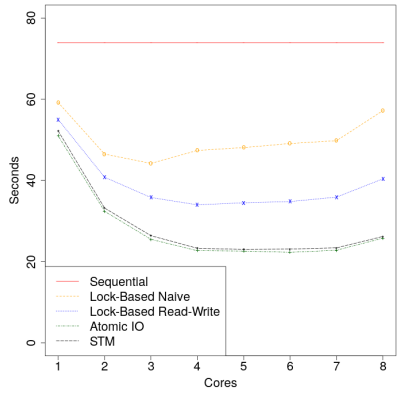
Comparing the performance and scaling to multiple cores of the STM and both Lock-Based implementations shows that the STM implementation significantly outperforms the Lock-Based ones and scales better to multiple cores. The Lock-Based implementations perform best with 3 and 4 cores respectively, and shows decreasing performance beyond 4 cores as can be seen in Figure 8.2. This is no surprise because the more cores, the more contention for the central lock. Thus, it is more likely that synchronisation is happening, ultimately resulting in reduced performance. This is not an issue in STM because no locks are taken in advance due to optimistic locking, where a log of changes is kept allowing the runtime to trigger a retry if conflicting changes are detected upon transacting.

Deleted: ,
Deleted: t
Deleted: the

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

Deleted: is slightly outperforming

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



Grid Size
101 x 101 (10,201)
151 x 151 (22,801) 212.0 (0.16) 201 x 201 (40,401) 382.0 (0.85) 251 x 251 (63,001) 608.0 (1.20)

Table 8.3: Performance comparison of Lock-Based Read-Write, Atomic IO and STM SIR implementations with varying grid sizes on 4 cores. Timings in seconds (lower is better), standard deviation in parentheses.

Figure 8.3: Performance comparison of Lock-Based Read-Write, Atomic IO and STM SIR implementations with varying grid sizes on 4 cores.

8.3.3 Varying Grid Size, Constant Cores

Deleted: g
Deleted: s
Deleted: c
Deleted: c
Deleted: constantly

In this experiment, we varied the grid size and used 4 cores constantly. The results are reported in Table 8.3 and plotted in Figure 8.3.

It is clear that the STM implementation outperforms the Lock-Based implementation by a substantial factor. However, the Atomic IO implementation outperforms the STM one again, where this time the difference is a bit more pronounced due to the higher workload of the experiments.

8.3.4 Retries

The retry ratio is extremely interesting when using STM, which indicates how many of the total STM actions had to be re-run. A high retry ratio shows that a lot of work is wasted on re-running STM actions due to many concurrent read

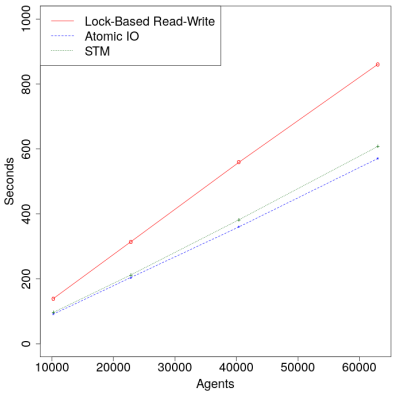
Deleted: Of very much interest when using STM is the retry ratio

Lock-Based Read-Write

Atomic IO

STM 96.5 (0.27)

139.0 (0.15)	91.1 (0.14)
314.0 (0.67)	204.0 (0.36)
559.0 (1.22)	360.0 (0.61)
861.0 (0.62)	571.0 (0.71)



CHAPTER 8.

CONCURRENCY IN ABS

125

Grid Size
51 x 51 (2,601) 101 x 101 (10,201) 151 x 151 (22,801) 201 x 201 (40,401) 251 x 251 (63,001)

Commits

Retries Ratio 0.0

0.0 0.0 0.0 0.0

2,601,000	1306

10,201,000	3712
22,801,000	8189
40,401,000	13285
63,001,000	21217

Table 8.4: Retry ratios of the SIR STM implementation with varying grid sizes on 4 cores.

Atomic IO STM

Cores

51x51

251x251

638.0 (8.24) 720.0 (1.70)

307.0 (1.12) 269.0 (1.05)

16	18.0 (0.21)
32	15.6 (0.07)
16	14.5 (0.03)
32	14.7 (0.17)

Table 8.5: Performance comparison of Atomic IO and STM SIR implementations on 16 and 32 cores on an Amazon EC2 m5ad.16xlarge instance. Timings in seconds (lower is better), standard deviations in parentheses.

and writes. Obviously, it is highly dependent on the read-write patterns of the implementation and indicates how well an STM approach is suitable for the problem at hand. We used the stm-stats [27] library to record statistics of commits, retries and the ratio. The results are reported in Table 8.4.

Independent of the number of agents we always have a retry ratio of 0. This indicates that this model is very well suited to STM, which is also directly reflected in the much better performance over the Lock-Based implementations. Obviously, this ratio stems from the fact, that in our implementation we have very few conditional writes, which happen only in cases when an agent changes from Susceptible to Infected or from Infected to Recovered.

8.3.5 Going Large Scale

Deleted: l

Deleted: s

To test how far we can scale up the number of cores in the best performing cases, Atomic IO and STM, we ran two experiments, 51x51 and 251x251, on an Amazon EC m5ad.16xlarge instance with 16 and 32 cores to see if we are running into decreasing returns. The results are reported in Table 8.5.

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

increased number of retries of the atomic modification operation, which obviously increases when the number of agents increases. The STM implementation performance on the other hand nearly stays constant on 16 and 32 cores in the 51x51 case. In both cases we measured a retry ratio of 0. Thus, we conclude that with 32 cores we become limited by the overhead of STM transactions [148], because the workload of an STM action in our SIR implementation is quite small. On the other hand, with a heavy load as in the 251x251 case, we see an increased performance with 32 cores.

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

8.3.6 Summary

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

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

8.4 Case Study II: Sugarscape

The second case study is the Sugarscape model as introduced in Chapter 2.1.2. In this case study we look into the potential for performance improvement in a model with much more complex agent behaviour and dramatically increased writes on the shared environment.

We implemented the Carrying Capacity (p. 30) section of Chapter II of the Sugarscape book [50]. In each step agents 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

Deleted: ,
Deleted: t
Deleted: on
Deleted: as
Deleted: to be not
Deleted: on
Deleted: case
Deleted: ,
Deleted:
Deleted: to have
Deleted: the
Deleted: s

level depending on the model parameters. This is in accordance with our results as we show in Appendix A 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 four directions N, W, S, E;
- Sugar growback: sugar grows back by 1 unit per step until the maximum capacity of a cell is reached;
- Agent population: initially 500 agents;
- Environment size: 50 x 50 cells with toroid boundaries wrapping around

  in both x and y dimensions.

  In this implementation (as in the full Chapter II of the book), no direct and

no synchronous agent interactions occur as we implemented them in Chapter 5. As in the SIR example, all agents interact with each other indirectly through the shared environment. This allows us to regard the implementation as a time-driven, parallel one where in each step, agents act conceptually at the same time.

8.4.1 Experiment Design

In this case study we compare the performance of four (4) implementations under varying numbers of CPU cores and agent numbers. The code of all implementations can be accessed freely from the code repository [176].

1. Sequential - This is the reference implementation, where all agents are run after another (including the environment). The environment is represented using an indexed array [109] and shared amongst the agents using a StateT Transformer.
2. Lock-Based - This is the same implementation as Sequential, but all agents are run concurrently within the IO Monad. The environment is also represented as an indexed array, but shared using a global reference between the agents that acquires and releases a lock when accessing it. Note that the semantics of Sugarscape do not support the implementation of either a read-write lock or an atomic modification approach as in the SIR model. In the SIR model, the agents write conditionally to their own cell, but this is not the case in Sugarscape. Here, the agents need a consistent view of the whole environment for the whole duration of an agent execution. This requirement is due to the fact that agents do not only write their own locations but also to other locations. If this is not handled correctly, data races happen and threads overwrite data from other threads, ultimately resulting in incorrect dynamics.
3. STM TVar - This is the same implementation as Sequential, but all agents are run concurrently within the STM Monad. The environment is also represented as an indexed array but shared using a TVar between the agents.

Deleted: ¶

Deleted: are happening

Formatted: Outline numbered + Level: 1 + Numbering Style: 1, 2, 3, … + Start at: 1 + Alignment: Left + Aligned at: 0,63 cm + Tab after: 1,27 cm + Indent at: 1,27 cm

Deleted: -

Deleted: which

Deleted: ,

Deleted: where

Deleted: ¶

Deleted: are happen- ing

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

Ordering the model specification requires shuffling agents before every step ([50], footnote 12 on page 26). In the Sequential approach we do this explicitly, but in the Lock-Based and both STM approaches we assume this happens automatically due to race conditions in concurrency. Thus, we arrive at an effectively shuffled processing of agents because we implicitly assume that the order of the agents is effectively random in every step. The important difference between the two approaches is that in the Sequential approach we have full control over this randomness, but in the STM this is not the case. This has the consequence that repeated runs with the same initial conditions might lead to slightly different results. This decision leaves the execution order of the agents ultimately to Haskell's runtime system and the underlying operating system. We are aware that by doing this, we make assumptions that the threads run as uniformly distributed. But such assumptions should not be made in concurrent programming. As a result, we can expect this fact to produce non-uniform distributions of agent runs, but we assumed that for this model this does not have a significant influence. In case of doubt, we could resort to shuffling the agents before running them in every step. This problem, where the influence of non-deterministic ordering on the correctness and results of ABS also has to be analysed, deserves in-depth research on its own. We introduce techniques allowing us to perform such analyses in Chapters 10 and 11 on property-based testing, but leave it for further research as this issue is beyond the focus of this thesis.

Deleted: T
Deleted: to shuffle
Deleted: to happen
Deleted: t
Deleted: not
Deleted: (fair)
Deleted: b
Deleted: s
Deleted: s
Deleted: ce
Deleted: also

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

Deleted: ,
Deleted: ,

The experiment setup is the same as in the SIR case study, with the same hardware (see Table 8.1), with measurements done under no additional work-

Cores 1
2
3
4
5
6
7
8

Sequential

Lock-Based

TVar

TArray 42.0 (2.20) 24.5 (1.07) 19.7 (1.05) 18.9 (0.58) 20.3 (0.87) 21.2 (1.49) 21.0 (0.41) 21.1 (0.64)

25.2 (0.36)	21.0 (0.12)	21.1 (0.25)
-	20.0 (0.12)	22.2 (0.21)
-	21.9 (0.19)	23.6 (0.12)
-	24.0 (0.17)	25.2 (0.16)
-	26.7 (0.17)	31.0 (0.24)
-	29.3 (0.57)	35.2 (0.12)
-	30.0 (0.12)	38.7 (0.42)
-	31.2 (0.29)	49.0 (0.41)

Table 8.6: Performance comparison of Sequential, Lock-Based, TVar and TArray Sugarscape implementations under varying cores with 50x50 environment and 500 initial agents. Timings in seconds (lower is better), standard deviation in parentheses.

Cores TVar TArray 1 0.00

2 0.02 3 0.04 4 0.06 5 0.07 6 0.09 7 0.10 8 0.11

Table 8.7: Retry ratio comparison (lower is better) of the TVar and TArray Sugarscape implementations under varying cores with 50x50 environment and 500 initial agents.

load using the microbenchmarking library Criterion [138, 139] as well. However, as the Sugarscape model is stepped using natural numbers we ran each measurement until t = 1000 and stepped it using Δt = 1. In the experiments we varied the number of agents as well as the number of cores when running concurrently. We checked the visual outputs and the dynamics and they look qualitatively the same as the reference Sequential. As in the SIR case study, a rigorous, statistical comparison of all implementations to investigate the effects of concurrency on the dynamics is quite involved and therefore beyond the focus of this paper. But, as a remedy we refer to the use of property-based testing, as shown in [182].

Deleted: ,

Deleted: b

8.4.2 Constant Agent Population

Deleted: a

Deleted: p

In this experiment we compare the performance of all implementations on varying numbers of cores. The results are reported in Table 8.6 and plotted in Figure 8.4.

0.00
1.04
2.15
3.20
4.06
5.02

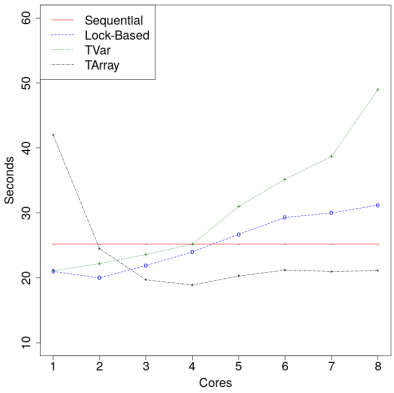
6.09
8.45

Figure 8.4: Performance comparison of Sequential, Lock-Based, TVar and TAr- ray Sugarscape implementations on varying cores with 50x50 environment and 500 initial agents.

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

The reason for the bad performance of TVar is that using a TVar to share the environment is a very inefficient choice: every write to a cell leads to a retry independent of whether the reading agent reads that changed cell or not, because the data structure cannot 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. The inefficiency of TVar is also reflected in the fact that the Lock-Based implementation outperforms it on all cores. The sweet spot is at 3 cores in both cases, after which decreasing performance is the result. This is due to very similar approaches because both operate on the whole environment instead of only the cells as TArray does. In case of the Lock-Based approach, the lock contention increases, whereas in the TVar approach, the retries start to dominate (see Table 8.7).

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



CHAPTER 8.

Agents 500 1,000 1,500 2,000 2,500

CONCURRENCY IN ABS

Deleted: one

Deleted:

Deleted: one

Deleted:

Deleted: in both cases

131

Sequential

Lock-Based

TVar

TArray 25.7 (0.42)

38.8 (1.43) 40.1 (0.25) 49.9 (0.82) 55.2 (1.04)

70.1 (0.41)	67.9 (0.13)	69.1 (0.34)
145.0 (0.11)	130.0 (0.28)	136.0 (0.16)
220.0 (0.14)	183.0 (0.83)	192.0 (0.73)
213.0 (0.69)	181.0 (0.84)	214.0 (0.53)
193.0 (0.16)	272.0 (0.81)	147.0 (0.32)

Table 8.8: Performance comparison of Sequential, Lock-Based, TVar and TArray Sugarscape implementations with varying agent numbers and 50x50 environment on 4 cores (except Sequential). Timings in seconds (lower is better), standard deviation in parentheses.

8.4.3 Scaling Up Agents

So far, we kept the initial number of agents at 500, which due to the model specification, quickly drops and stabilises around 200 due to the carrying capacity of the environment as can be seen in Figure ?? and which is also described in the book [50] section Carrying Capacity (p. 30).

We now measure the performance of our approaches under an increased number of agents. For this we slightly change the implementation: when an agent dies it always spawns a new one, which is inspired by the ageing and birthing feature of Chapter III in the book [50]. This ensures that we keep the number of agents roughly constant (it 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 also demonstrates the ability to terminate and fork threads dynamically during the simulation.

Except for the Sequential approach, we ran all experiments with 4 cores. We looked into the performance of 500, 1,000, 1,500, 2,000 and 2,500 (maximum possible capacity of the 50x50 environment). The results are reported in Table 8.8 and plotted in Figure 8.5.

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

What seems to be very surprising is that in the Sequential and TVar cases the performance with 2,500 agents is better than the one with 2,000 agents. The reason for this is that in the case of 2,500 agents, an agent can't move anywhere because all cells are already occupied. In this case the agent will not rank the cells in order of their payoff

Deleted: u

Commented [KJ2]: Don't forget to look this up!

Formatted: Highlight

Deleted: always

Deleted: also

Deleted: smother

Deleted: won't

(max sugar) as to where to move, but just stays where it is. We hypothesize that due to Haskell's laziness the agents never actually look at the content of the cells, but only at the number, which means that the cells themselves are never evaluated and this further increases performance. Better performance is seen in Sequential and TVar, because both exploit laziness. In the case of the Lock-Based approach we still arrive at a lower performance level because the limiting factor is the unconditional locks. In the case of the TArray approach we also arrive at a lower performance because it seems

Figure 8.5: Performance comparison of Sequential, Lock-Based, TVar and TArray Sugarscape implementations with varying agent numbers and 50x50 environment on 4 cores (except Sequential).

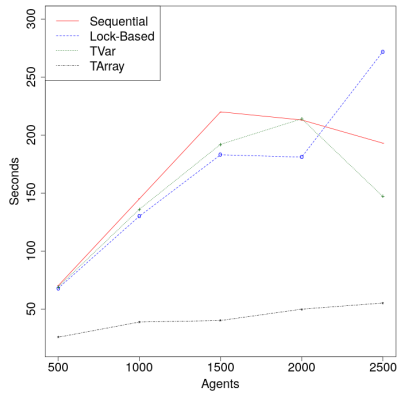
that STM perform reads on the neighbouring cells which are not subject to lazy evaluation.

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

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

8.4.4 Going Large Scale

To test how far we can scale up the number of cores in the TArray case, we ran the two experiments, carrying capacity (500 agents) and rebirthing (2500 agents), on an Amazon EC m5ad.16xlarge instance with 16, 32 and 64 cores to see if we run into decreasing returns. The results are reported in Table 8.9.



Deleted: to move to
Deleted: actually
Deleted: in this case
Deleted: which
Deleted: This leads to the
Deleted: b
Deleted: case of
Deleted: are
Deleted: ,
Deleted: where t
Deleted: l
Deleted: s

Cores Carrying Capacity Rebirthing

133

11.9 (0.21)
12.8 (0.29)
14.6 (0.09)

16 32 64

46.6 (0.07) 76.4 (0.01) 99.1 (0.01)

Table 8.9: Sugarscape TArray performance on 16, 32 and 64 cores an Amazon EC m5ad.16xlarge instance. Timings in seconds (lower is better). Retry ratios in parentheses.

Unlike in the SIR model, Sugarscapes STM TArray implementation does not scale up beyond 16 cores. We attribute this to a mix of retries and Amdahl's law. As retries are much more expensive in the case of Sugarscape compared to SIR, even a small increase in the retry ratio (see Table 8.7), leads to reduced performance. On the other hand, although the retry ratio decreases as the number of cores increases, the ratio of parallelisable work diminishes and we get bound by the sequential part of the program.

8.4.5 Comparison with Other Approaches

The paper [115] reports a performance of 2,000 steps per second on a GPU on a 128x128 grid. Our best performing implementation, TArray with 500 rebirthing agents, arrives at a performance of 39 steps per second and is therefore clearly slower. However, the very high performance on the GPU does not concern us here as it follows a very different approach than ours. We focus on speeding up implementations on the CPU as directly as possible without locking overhead. When following a GPU approach, one needs to map the model to the GPU which is a delicate and non-trivial matter. With our approach we show that speed up with concurrency is very possible without the low-level locking details or the need to map to GPU. Additionally, some features like bilateral trading between agents, where a pair of agents needs to come to a conclusion over multiple synchronous steps, is difficult to implement on a GPU whereas this should be not as hard using STM.

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

Deleted: o

Deleted: a

Deleted: lso

Deleted: ,

8.4.6 Summary

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

Furthermore, this case study demonstrated that the selection of the right transactional data structure is of fundamental importance when using STM. Selecting the right transactional data structure is highly model-specific and can lead to dramatically different performance results. In this case study, the TArray performed best due to many writes but in the SIR case study a TVar showed good enough results due to the very low number of writes. When not carefully selecting the right transactional data structure, which supports fine-grained concurrency, a lock-based implementation might perform as well or even outperform the STM approach as can be seen when using the TVar.

Although the TArray is the better transactional data structure overall, it might come with an overhead, performing worse on low number of cores than a TVar, Lock-Based or even Sequential approach, as seen with TArray on 1 core. However, it has the benefit of quickly scaling up to multiple cores. Depending on the transactional data structure, scaling up to multiple cores hits a limit at some point. In the case of the TVar the best performance is reached with 3 cores. With the TArray we reached this limit around 16 cores.

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

8.5 Discussion

In this chapter we have shown how to apply concurrency to monadic ABS to gain a substantial speedup. We developed a novel approach, using STM, which to our best knowledge has not been discussed systematically in the context of ABS so far. This new approach outperforms a traditional lock-based implementation running in the IO Monad and guarantees that the differences between runs with same initial conditions stem

Deleted:

Deleted:

Deleted: would

Deleted: ,

Deleted: which

Deleted: ly

from the non-determinism of STM, but nothing else. The latter point can't be guaranteed by the lock-based approach as it runs in the IO Monad, which allows literally anything from reading from a file, to launching a missile. Additionally, with STM, concurrency becomes more of a control-flow oriented concern. So, using STM allows us to treat the concurrent problem within an agent as a data-flow oriented one, without cluttering the model code with operational details of concurrency. This gives strong evidence that STM should be favoured over lock-based approaches in general for implementing concurrent ABS in Haskell.

Also, STM primitives map nicely to ABS concepts. When having a shared environment, it is natural to use either TVar or TArray, depending on the environment's nature. What is more, the TChan primitive exists, which can be seen as a persistent message box for agents, underlining the message-oriented approach found in many agent-based models [2, 202]. Moreover, TChan offers a broadcast transactional channel, which supports broadcasting to listeners and maps nicely to a proactive environment or a central auctioneer, which agents need to synchronize. The benefits of these natural mappings are that using STM takes a big portion of the burden from the modeller, as one can think in STM primitives instead of low level locks and concurrent operational details.

We assumed that concurrent execution has no qualitative influence on the dynamics. Although repeated runs with same initial conditions might lead to different results due to non-determinism, the dynamics still follow the same distribution as the one from the sequential implementation. To verify this, we can make use the techniques of property-based testing as shown in Chapters 10 and 11 of this thesis, but we will leave it for further research.

The next step would be to add synchronous agent interactions as they occur in the Sugarscape model and use cases of mating, trading and lending. We have started to do work on this already and could implement one-directional agent interactions as well, as they occur in the disease transmission, payback of loans and notification of inheritance upon the death of a parent agent. We use the TQueue primitive to emulate the behaviour of mailboxes through which agents can post events to each other. The result is promising, but needs more investigation. We have also started looking into synchronous agent interactions using STM which is a lot trickier and is very susceptible to deadlocks, which are still possible in STM! We have yet to prove how to implement reliable synchronous agent interactions without deadlocks in STM. It might be very well the case that a truly concurrent approach is doomed due to the following [123] (Chapter 10. Software Transactional Memory, "What Can We Not Do with STM?"): "In general, the class of operations that STM cannot express are those that involve multi-way communication between threads. The simplest example is a synchronous channel, in which both the reader and the writer must be present simultaneously for the operation to go ahead. We cannot implement this in STM, at least compositionally [..]: the operations need to block and have a visible effect — advertise that there is a blocked thread — simultaneously.".

Furthermore, STM is not fair because all threads, which block on a transactional primitive, have to be woken up. Thus, a FIFO guarantee cannot be given. We hypothesise that for most models where the STM approach is applicable, this has no qualitative

Deleted: Further
Deleted: ,
Deleted: s
Deleted: using
Deleted: Also
Deleted: there exists
Deleted: Also
Deleted: which
Deleted: upon
Deleted: follow
Deleted: also
Deleted: ,
Deleted: t

influence on the dynamics as agents are assumed to act conceptually at the same time and no fairness is needed. We will leave the test of this hypothesis for future research.

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

Deleted: , for example

We did not look into applying distributed computation to our approach. One direction to follow would be to use the Cloud Haskell library, which is very similar to the concurrency model in Erlang. We will leave this for further research as it is beyond the scope of this thesis.

Deleted: n't