HLogo: a Parallel Haskell Variant of NetLogo

Nikolaos Bezirgiannis¹, I. S. W. B. Prasetya², Ilias Sakellariou³

¹Centrum Wiskunde & Informatica (CWI), P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
²Dept. of Inf. and Comp. Sciences, Utrecht University, P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
³Dept. of Applied Informatics, University of Macedonia, 156 Egnatia Str. 54636 Thessaloniki, Greece

n.bezirgiannis@cwi.nl, s.w.b.prasetya@uu.nl, iliass@uom.edu.gr

Keywords: Agent-based Modeling, Agent-based Simulation, Concurrent Agent-based Simulation, Concurrent NetLogo.

Abstract: Agent-based Modeling (ABM) has become quite popular to the simulation community for its usability and wide area of applicability. However, speed is not usually a trait that ABM tools are characterized of attaining. This paper presents HLogo, a parallel variant of the NetLogo ABM framework, that seeks to increase the performance of ABM by utilizing Software Transactional Memory and multi-core CPUs, all the while maintaining the user friendliness of NetLogo. HLogo is implemented as a Domain Specific Language embedded in the functional language Haskell, which means that it also inherits Haskell’s features, such as its static typing.

1 INTRODUCTION

Agent-based Modeling (ABM) is a computer simulation technique that uses intelligent agents as its building blocks. ABM has gained traction lately, especially for its ease of use and broad applicability, e.g. in social sciences (Epstein and Axtell, 1996), ecology (Grimm et al., 2005), biology (Pogson et al., 2006), and physics (Wilensky, 2003). ABM offers the right level of abstractions to construct large agent populations that can exhibit interesting (from a simulation perspective) emergent patterns. This success of ABM sprang off numerous frameworks to alleviate the development of agent-based models (Tobias and Hofmann, 2004; Railsback et al., 2006; Castle and Crooks, 2006). While research has been focused on the methodology (Salamon, 2011), ease of use (Wilkerson-Jerde and Wilensky, 2010), portability (Grimm et al., 2006) and expressiveness (Sakellariou et al., 2008) of agent-based models, little has been done in improving the performance of available ABM frameworks (Riley and Riley, 2003).

The execution performance of ABM limits the agent population size and increases simulation time, which may impact the emergence of sought-after phenomena. To this end, we propose in this paper a new ABM framework called HLogo, that strives to speed up the simulation execution by harvesting the available parallelism of the ubiquitous, modern multi-core CPUs. The framework’s language & engine is implemented entirely in Haskell and is inspired much for its user-friendliness by NetLogo (Wilensky, 1999), arguably one of the most well-known and widely-used ABM framework. Unlike NetLogo, our framework offers three unique features which also constitute the main contribution of this paper:

- agents inside the simulation framework can run concurrently by utilizing a technology called Software Transactional Memory (STM) (Shavit and Touitou, 1995). Coupled with Haskell’s lightweight (green) threads, the overall ABM execution enjoys significant benefits from multi-core parallelism.
- the framework is embedded as a Domain Specific Language (eDSL), which comes with the advantage of ease of language, framework, and program extensibility (via modules and plugins), but limits the syntax to that of the host language Haskell.
- the HLogo language is statically typed with type inference, which strengthens the ABM program safety, yet not burden the user with writing type annotations.

The rest of this paper is organised as follows. Section 2 introduces the main concepts of ABM and their realization in the HLogo language. Section 3 explains how HLogo implements multi-core execution. In Section 4 we show the performance and scalability results of our benchmarking, which also includes a comparison with NetLogo. In Section 5 we visit related work and how others have addressed the same problem. Fi-
nally, Section 6 maps out future research and concludes.

2 HLOGO’S LANGUAGE

2.1 Main ABM Concepts

An agent-based model is a system of multiple agents which are stateful entities, capable of carrying out actions and interact with each other. A simulation run is the execution of such an agent model, where the agents ‘drive’ the computation. Following NetLogo’s conventions, agents live in a tiled 2D and drawable environment, which we will refer to as the canvas. There are three types of agents: patches, turtles\(^1\), and links. Patches are stationary, occupying a single tile on the canvas; they are statically created at the start of a simulation run and cannot be later destroyed. Turtles can be programmatically created & destroyed, and move around the canvas. Links are agents with a dynamic lifespan as well; each representing a connection between two turtles.

Each type of agent has a different set of mutable attributes that form its state; e.g. turtles have attributes xcor and ycor to represent their position on the canvas, patches have attribute pcolor to represent the color of the tile, etc. Besides standard attributes, the user may introduce new attributes to the agents by defining ‘breeds’. A breed extending an agent-type (turtle or link) with custom attributes, similar to the object-oriented concept of final class (cannot be further subclassed). For example, the NetLogo code below declares a turtle-breed called cows, with age and hunger as extra attributes:

```netlogo
breed [cows cow]
cows-own [age hunger]
```

The declaration implicitly creates a global variable cows that contains all currently alive cows, as well as getters and setters (such as hunger and set_hunger) to get and set the attributes of a cow.

An agent can command any other agent to execute some code (e.g. for changing its state or interacting with each other) through the ask operation. For example, if \( T \) is a set of cows, the NetLogo statement `ask T [set color brown set_hunger 0]` commands every cow in \( T \) to change the color of the patch it is currently on to brown (e.g. to represent that the cow has eaten all the grass in the patch), and then set its hunger attribute to 0 (representing ‘not hungry’). Two other operations are provided: ‘of’ queries a set of agents for their values, e.g. \([\text{xcor}] \) of turtles, and ‘with’ filters an agent-set based on a given predicate, e.g. `patches with [color = red]`. Besides built-in commands (e.g. set), an agent can execute custom commands by user-defined procedures. Lastly, the built-in variables self and myself refer to the currently executing agent (similar to ‘this’ in OO), and the parent caller that asked this agent, respectively.

2.2 Haskell Embedding

Both NetLogo and HLogo are Domain Specific Languages (DSLs) to describe simulations. Whereas NetLogo is a native DSL, HLogo is embedded (eDSL) inside a general purpose language\(^2\), namely Haskell, which is a lazy-by-default functional language. An eDSL is easier to build, as there is no need to create a separate compiler or interpreter. Furthermore, an eDSL inherits its host language’s features; e.g. (for Haskell) the support for higher-order functions and overloading make Haskell particularly attractive for embedding DSLs (Bjøsset al., 1998; Elliott, 2003; Peterson and Hager, 1999).

With the HLogo’s eDSL, the simulation user can benefit from the Haskell’s module system by organizing the agent program into separate modules, or importing and reusing code from the already vast collection of open-source Haskell libraries (Hackage). NetLogo, on the other hand, currently (as of version 5.2.1) lacks a module system.

A second reason for choosing the eDSL approach, is that HLogo can be more easily extended with new language constructs, and plugins targeting the simulation engine (e.g. visualization). Still, embedding does mean that the syntax of HLogo is constrained by that of Haskell. For example, in NetLogo binary operators have higher precedence than function application; e.g. `print 1+3`, whereas in Haskell the precedence is reversed; so, we have to write `print (1+3)`.

Typing. Other than the syntax, HLogo inherits Haskell’s static typing: all HLogo expressions are statically typed, which is in contrast to NetLogo’s dynamic typing — NetLogo performs very minimal type checking. For example, in HLogo the type error in the expression `1+non_number` will be detected at

\(^1\)Historically, the concept of ‘turtle’ has its origin in the educational programming language Logo, which was inspirational for NetLogo.

\(^2\)A native DSL has a dedicated parser and a compiler or interpreter to execute its code; e.g. NetLogo compiles its code to Scala. An eDSL is a DSL embedded inside another language (host). It tries to mimic a native DSL by providing its language constructs in terms of the constructs of the host.
compile time, whereas NetLogo will only detect this later at runtime. To this degree, we are allowed to say that HLogo provides more safety for agent model programs.

An additional significant advantage of Haskell’s typing is that HLogo can type-check not just simple arithmetic expressions, but more elaborate statements that contain some agent context: almost all built-in Logo commands can be executed in a restricted agent context (self, myself); e.g. the command `forward N` may only be executed by a turtle agent (and moves the turtle N units forward on the canvas). In the example that follows, we erroneously ‘ask’ a patch to move `forward` (they are stationary) and `die` (cannot be destroyed):

```
%% NetLogo: yields error only later at runtime
ask patch x y [forward 3
    die]

-- HLogo: program does not type-check
ask (atomic (do forward 3
    die)) =<< patch x y
```

Type-checking of such agent commands is achieved through Haskell’s typeclasses, a similar concept to OO interfaces used for ad-hoc polymorphism. Specifically, every built-in HLogo command is typed with its expected agent context, e.g. `die` takes the Haskell type

```
die :: ∀a.∀m.TurtleLink a ⇒ Logo a m ()
```

where `TurtleLink` is a typeclass that points to either a turtle or a link, `a` is the self context and `m` is any self context.

Haskell’s strong type system also comes with Hindley-Milner type inference (Milner, 1978), which makes type annotations optional. This overall provides type safety to the user, without the burden of annotating the code with type signatures. As can easily be witnessed from the illustrated example above, no type annotations were necessary for the HLogo code since the underlying Haskell compiler can derive the types of HLogo expressions — `die` expects a `TurtleLink`, `forward` expects turtle-only, so by implication Haskell expects a turtle as the agent’s self context.

**Monads.** Since agents are stateful, what they do may also have side effects on their own or each other’s states. Haskell is by intent a purely functional language that has no natural concept of side effect. Side effect is brought into the language as instances of so-called `monad` (Peyton Jones and Wadler, 1993), which is a generic concept representing a category of things sharing a set of algebraic laws, e.g. associativity. A monadic action `e` is an expression of type `M t` where `M` is a type that specifies the structure of the monad `e` operates on (which could be a state), and `t` is the type of the ‘return value’ of this action. So, if `e` is a command to a turtle `a`, then `M` is a type describing `a`’s state structure (and that of the patch `a` is on).

Monadic actions can be *sequentially composed* with the `do` notation. E.g., suppose `c` is a monadic action, then the expression: `do { z <- c; return (z+1) }` is a new monadic action, that first executes `c`, captures its return value in `z`, then returns `z + 1`. More generally, the expression `do{e₁ : e₂ : ...}` will evaluate the expressions `e₁, e₂ ...` in the given order. When `e₁, e₂, ...` are written vertically, and starts at the same column, we can drop the use of delimiter “`,`”, and `;”` to get a cleaner syntax. A `do`-sequence that does not explicitly specify a return as the last expression implicitly returns whatever the last expression returns. For example, `count` is a monadic action that returns given agentset’s size:

```
do
    p <- patches
    count p
```

There is also the `e =<< d` operator that can be used to directly pipe the value returned by the monadic action `d` as an input for `e`. So, the above code can also be written more compactly as `count =<< patches`. In HLogo, any agent action that has side-effects is expressed as a monadic action.

**Procedures.** Defining a procedure in NetLogo is done through the `to` ... `end` syntax. E.g. the code below defines the `move[p]` procedure, which will turn all cows 5 degrees to the right, then move them `p` points forward:

```
to move [p]
    ask cows [rt 5 fd p]
end
```

In Haskell, the same definition is achieved by a top-level function bound to its corresponding right-hand side monadic action:

```
move p = do
    a <- cows
    ask {atomic (do { rt 5; fd p })} a
```

Or more compactly as:

```
move p = do
    ask {atomic (do { rt 5; fd p })} =<< cows
```

The built-in command `atomic` controls the concurrency and will be detailed in Section 3. Haskell (and thus HLogo) does not in principle support polyvariadic procedures of NetLogo (procedures that allow variable number of arguments to be passed to
them), but there exist Haskell packages that can simulate this feature, e.g. HList (Kiselyov et al., 2004) or Liquid Haskell (Vazou et al., 2014). Conversely, a Haskell function unifies the concepts of the NetLogo procedure and the NetLogo reporter (procedure with return value), and, moreover, can support anonymous reporters via lambda abstractions (a current limitation of NetLogo as of version 5.2.1).

Extending agents with new breeds or attributes requires the user to define a new datatype, together with a set of getter/setter functions. To avoid having to write such boilerplate code, we employ Template Haskell (Sheard and Jones, 2002), a compile-time meta-programming technique that will generate the needed code. The following Haskell code is a preamble of an example HLogo model that showcases the use of Template Haskell macros:

```haskell
-- HLogo eDSL is a library
import Language.Logo

-- generates: cows,cows_here,...
breeds ["cows", "cow"]

-- generates getter/setters: energy
b Breeds_own "cows" ["energy"]

-- generates program's entrypoint
run ["setup", "go"]
```

Nearly the complete set of NetLogo’s standard library has been ported. A rudimentary support for visualization is present: the command snapshot can be called at any place in HLogo code to save an image of the current simulation’s 2D canvas to a fresh postscript image. The code in Listing 1 shows an example HLogo model of cows (turtles) moving & eating grass (patches), with its snapshot image in Fig. 1. Live visualization, as offered by the NetLogo platform, is part of future work.

```
setup = do
  ask (do
    c <- one_of [green, brown]
    atomic (set_pcolor c)
    ) =<< patches
  cs <- create_cows 50
  ask (do
    x <- random_xcor
    y <- random_ycor
    atomic (do
      set_color white
      set_energy 50
      setxy x y
    ) ) cs
  reset_ticks

go = forever (do
  t <- ticks
  when (t > 10) stop
  ask (do
    move
    eat_grass) =<< cows
    snapshot
tick)

move = do
  r <- random 50
  atomic (do
    right r
    forward 1)

eat_grass = atomic (do
  c <- pcolor
  when (c == green) (do
    set_pcolor brown
    e <- energy
    set_energy (e+30))
```

Listing 1: An example of agent model in HLogo. Cows-turtles move around and eat grass-patches to gain energy.

3 HLOGO’S EXECUTION

The HLogo user arranges the agent model in a number of Haskell source files which are then compiled via a Haskell compiler into native code and executed to start the simulation run. A NetLogo model is instead parsed and translated into intermediate bytecode which gets interpreted by a JVM; however, nowadays JVM employs regularly Just-In-Time (JIT) compilation to native code. Both HLogo and NetLogo simulation engines use similar data-structures to store the agents: 2-dimensional array for patches, map tree for turtles/links. Their principal difference is on how their commanding operators (ask/of/with) are executed: NetLogo calls each agent in turn (sequentially), and the called agent runs the body of commands to completion. NetLogo also provides the al-

![Figure 1: An example of HLogo visualization output. White triangles represent cows, patches are (eaten) grass.](image-url)
Software Transactional Memory (STM) is a concurrency control mechanism that allows concurrent processes to transparently yet safely operate on common resources. It departs from the common locking mechanism by avoiding locks altogether. STM’s concurrency relies on the so-called transactions. A transaction is a sequence of reads and writes to a set of transactional variables, or TVars for short, which point to actual places in a shared memory; the transaction will not write to those places directly. TVars are normally shared between transactions. A transaction is virtually atomic. That is, its intermediate changes on its TVars cannot be witnessed by other transactions. The idea originates from the field of Distributed Databases; indeed, people who are familiar with SQL databases can view STM transactions as the usual SQL atomic transactions albeit with a bigger focus on concurrency. Historically, Transactional Memory was introduced a long time ago as a novel extension to Lisp with suitable hardware modifications to enable concurrency (Knight, 1986). The idea was later refined in (Shavit and Touitou, 1995), which also coined the term Software Transactional Memory by realizing a purely in-software implementation of the approach. Recently, STM programs can be further accelerated through hardware instruction-set extensions, e.g. with Transactional Synchronization Extensions (TSX) of Intel® Skylake processor.

Each transaction $\tau$ can in principle be assigned to a separate thread that keeps a separate log of reads and writes to the transaction’s TVars. These writes are not committed yet. At the end of the transaction, the thread checks for possible inconsistencies in the log. If one is found, because another transaction has in the mean time updated one of the TVars, $\tau$ is aborted, and later retried again. Importantly, aborted transactions have no side effect. If there are no inconsistencies, the transaction is said to be successful, and the changes made on the TVars are committed (executed). Internally, the orchestration of a commit is realized by automatically putting locks in proper places in the shared memory, but this involved process is hidden from the programmers, thus effectively alleviating concurrent programming. For our case this is important, since we want to maintain NetLogo’s user friendliness. Using STM does incur overhead, but still, considerable speedup is observed (Perfumo et al., 2008).

Ultimately, transactions are run by threads for concurrent execution. There are several options on how to do this. The obvious choice is to run every transaction on its own thread. Haskell provides green (lightweight) threads. These are virtual threads managed by a virtual machine (or by a language’s advanced runtime-system), as opposed to native threads managed by the Operating System. Green threads have a smaller memory footprint, and faster thread activation and synchronization. A large number (thousands; even millions) of such threads can be spawn without running out of memory. Haskell runtime employs an M:N threading model, where M lightweight threads are automatically mapped to N kernel ( heavyweight) threads for multi-core parallelism. While this maximizes the parallelism, the number of actual CPU cores is usually much less than the number of agents. The above solution would lead to performance degradation. Subsection 3.2 discusses our solution.

The choice of Haskell was made for its strong and quality-standard STM library (Discolo et al., 2006), coupled with multi-core-enabled lightweight threads. Moreover, it is important that the effects of a transaction can be rollbacked, in case the transaction fails to commit: the type system of Haskell guarantees that STM effects cannot be intermixed with other effects which cannot be rollbacked (e.g. doing IO by writing to a file), since they belong to two different monads (e.g. STM monad vs IO monad).

3.2 Parallelizing HLogo

During execution, the ask/of/with operators ‘drive’ the computation of simulation. In HLogo, the ask operator is implemented as a function that takes as input a block of commands and a set of agents. First, the function splits the given agentset and the current random number generator (RNG) into equal pieces — the splitting is specialized for the agentset type (array or treemap) and we make use of a high-quality splittable treeshif RNG library (Claessen and Pałka, 2013). Next, the function spawns as many lightweight threads as there are CPU cores. Each spawned thread will execute the block of commands for each agent in its agentset slice in a modified (self,myself) agent context. A Haskell pseudocode of the above description is given in Listing 2.

This simple motif of ‘divide and conquer’ is used to parallelize the other two operators (of/with), the only difference being that the caller will collect/fil-
ask cmds agentset = do
caller <- self
r <- currentRNG
cores <- getNumCapabilities
-- splits agentset in CPU-core slices
let splitted = split agentset r cores
threads <- forM splitted (\(slice, rng)\) ->
    forkIO (do
        setRNG rng
        forM slice (\callee\) ->
            -- callee = > self, caller = > myself
            executeIn (callee, caller) cmds)
    )
forM threads wait -- caller blocks

Listing 2: Pseudocode for HLogo 'ask' implementation.

ter back the results of the block. All ask/of/with operators will make the caller block until the spawned 'worker' threads have finished (which implies that all the workload of that operator has completed); there is also a non-blocking variant of ask (named ask_async) where the caller immediately continues. Again all these three operators support nesting: it is perfectly allowed to call an ask inside an ask, ask inside of, or any other operator thereof; for example the agent program ask(ask eat_grass =<< cows_here)=<<patches (i) will have maximum (N*N)(+1 for the caller) threads running; the Haskell runtime system will automatically load-balance the threads to the CPU cores if for example there are some patches with less or no cows sitting on.

A Haskell thread expects to execute some code with IO effects (an IO monadic action); as such the IO effects will be applied in parallel to other threads. In case of HLogo, if we allow the agent commands to operate (apply their effects) in the IO monad, race conditions might happen between the agents. Consider an example HLogo procedure:

eat_grass = do
g <- atomic grass
    atomic (when (g > 30) (do
        set_grass (g - 30)
        e <- energy
        set_energy (e + 30)))

If we allow the above agent commands (getters/setters of grass and energy) in the body of the procedure to operate in IO, two race conditions may happen: a) two cows eat grass from the same patch, but the patch grass level is decreased only once; b) at another point in the program, an agent 'ask's to (destructively) modify the energy of a currently-eating cow.

Instead, what HLogo actually does is store all agent attributes into TVars and restrict basic agent commands (all standard library and getters/setters), to be allowed only inside an STM transaction; in other words, the code at (i) would not even type-check. Multiple agent commands can then be monadically sequenced as usual into a bigger STM transaction block. We extend the language with the command atomic which given a transaction block will try to 'run' it: when the atomic succeeds, it means its effects have been committed as a whole to the outside (IO) world, and will not be again roll-backed. The type of atomic is atomic :: STM a -> IO a, which reads as a function that lifts an STM transaction (monadic action) to an IO monadic action. To fix & type-check the HLogo code at (i) we simply surround the whole eat_grass in an 'atomic' block: ask (atomic eat_grass)=<< cows_here) =<< patches, which is parallel, and race-condition free.

Does this mean that the user should create as large transaction blocks as possible and merely surround them with a single 'atomic'? Not exactly, since larger transactions can hurt performance, because of larger kept STM logs and potentially more rollbacks caused by other competitor-threads trying to commit their own, related transactions. With HLogo, it is solely left to the user to decide if the whole transaction should be atomic or if it is safe to break it into smaller atomic blocks. As an example, the following code albeit faster, does not avoid race-condition (a) (but race-condition (b) is avoided):

eat_grass = do
g <- atomic grass
    atomic (when (g > 30) (do
        set_grass (g - 30)
        e <- energy
        set_energy (e + 30)))

Despite the gain in parallelism, STM is not a 'silver bullet' to all the problems that are encountered in a parallel setting. Any executed STM transaction is inherently non-deterministic; in the simplest case, two simultaneous threads competing to modify the same TVar will not commit always in the same order: there may be program runs where thread 1 commits before thread 2 and vice versa. As a consequence, this turns HLogo simulations non-reproducible, but still consistent with respect to race-conditions. On the bright side, HLogo's engine guarantees that on 1-core configurations and with no use of ask_async, the simulation of the agent model is reproducible.

4 EXPERIMENTAL EVALUATION

To compare the performance of HLogo to that of NetLogo, we ran the following benchmarks:
1. The benchmark *Redblue* has *N* turtles living on a 100 × 100 torus-shaped canvas. They move forward one step on every tick. If they are on a red patch, they also turn left by 30 degrees. If they are on a blue patch, they turn right by 30 degrees. This is simulated for 1000 ticks. In this benchmark, the agents never write to the same \( TVar \), and therefore the transactions never need to roll back.

2. The benchmark *Cows* has *N* cows living on a 100 × 100 torus-shaped canvas. They move around and eat grass. If the grass on a patch is consumed by a cow, after some time it will re-grow. This is simulated for 1000 ticks. In this benchmark, cows compete for the grass, so some transactions may conflict and have to roll back.

The benchmarks are run on a system provided by the SURF foundation with 16 cores Intel© Xeon E5-2698, 128GB RAM, and Hyper-Threading disabled. The OS Ubuntu 14.04 64bit was installed with The Glorious Glasgow Haskell Compiler version 7.10.3, NetLogo 5.2.1 running Java-8 version OpenJDK 1.8.0 72.

We benchmarked configurations with varying number of cores (1,2,4,8,16), and varying the problem size (*N*). We executed 20 simulation runs for each such configuration and computed the average runtime and resident memory; the results are shown in Table 1 (plotted in Fig. 2 and Fig. 3). Note that the NetLogo runtime includes only the actual execution of the model, not including the model parsing, compiling and firing up the JVM. We can clearly witness speed gain in HLogo, the more we increase the number of cores; the performance scalability is retained. The 16-core HLogo configuration manages to be at its best 87% faster than its NetLogo counterpart. Even with two cores and while using much less memory (74% less memory than NetLogo), HLogo manages to match or surpass the speed of NetLogo, which is a positive thing considering the fact that STM concurrency incurs a certain overhead.

The HLogo eDSL implementation, the HLogo/Netlogo benchmarks and examples are open-source software situated at http://github.com/bezirg/hlogo.

5 RELATED WORK

NetLogo (Wilensky, 1999) is one of the de-facto agent-based modeling framework. Models are written in the dynamically-typed NetLogo language, which is a dialect of the educational programming language Logo. Internally, NetLogo is implemented in the Scala programming language. A Scala-written compiler translates NetLogo code to Java bytecode to be later run in a Java Virtual Machine (JVM). NetLogo comes with a GUI to visualize simulation results, and a rich collection of predefined models, dealing with aspects of Biology, Computer Science, Earth Science, Social Science, Chemistry and others. ReLogo (North et al., 2013) is a NetLogo clone embedded as a DSL in Groovy (an OO language running in JVM). As is the case with NetLogo, ReLogo is single-threaded and comes with a rich GUI. Although Groovy version 2.0 and onwards introduced optional static typing, the ReLogo language cannot type-check many of its expressions: agentsets are untyped, and ask/of/with closures cannot track the type information of their context (self,myself). In such cases, the simulation user has to either resort to typecasting or turn off Groovy’s
Table 1: Benchmarks results of NetLogo against HLogo. N is the agent model problem size.

<table>
<thead>
<tr>
<th></th>
<th>runtime (sec)</th>
<th>memory usage (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NetLogo</td>
<td>HLogo, #cores</td>
</tr>
<tr>
<td>N</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Redblue</td>
<td>1000</td>
<td>0.41</td>
</tr>
<tr>
<td></td>
<td>2500</td>
<td>0.95</td>
</tr>
<tr>
<td></td>
<td>5000</td>
<td>1.64</td>
</tr>
<tr>
<td></td>
<td>10000</td>
<td>3.08</td>
</tr>
<tr>
<td></td>
<td>20000</td>
<td>6.02</td>
</tr>
<tr>
<td></td>
<td>30000</td>
<td>9.02</td>
</tr>
<tr>
<td>Cows</td>
<td>100</td>
<td>4.28</td>
</tr>
<tr>
<td></td>
<td>250</td>
<td>4.63</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>5.07</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>5.76</td>
</tr>
<tr>
<td></td>
<td>2000</td>
<td>6.42</td>
</tr>
<tr>
<td></td>
<td>3000</td>
<td>7.02</td>
</tr>
</tbody>
</table>

Both NetLogo and ReLogo support parameter sweeping (Koehler et al., 2005), for running multiple instances of the same model in parallel while varying the model's input parameters (e.g. initial random seed); however, this method is orthogonal to HLogo's parallelism and could as well be combined with HLogo: the purpose of parameter sweeping is to exploit multi-core by replicating the model and running different instances of it, whereas HLogo tries to inject parallelism inside a single instance of a simulation run. The latter case is crucial for large models or time-critical simulations where any performance gain in a single run is desirable.

To the best of our knowledge, the work described here is the first to apply Software Transactional Memory in Agent-Based Modeling. Other scientists in the field, however, have priorly investigated to speed up ABM execution using various parallel techniques: The work described in (Logan and Theodoropoulos, 2001) proposes to execute Agent-based systems through Distributed Discrete-Event Simulation. The key problem as they state is the decomposition of the environment which leads to the problem of fair load balancing of the distributed machines. (Riley and Riley, 2003) propose another Distributed Agent Simulation Environment called SPADES. SPADES tries to address the concerns of Artificial Intelligence when designing agent systems, while having distributed execution and reproducibility of results. (Massaioli et al., 2005) uses another parallelization technology, called OpenMP, to speed up the execution of agent-based models. However, the technique restricts the implementation of ABM frameworks to only that which provide an OpenMP implementation, i.e. C, C++, Fortran. Also, it adds the requirement to the simulation user to annotate simulation code with extra OpenMP pragmas, which is rather discouraging. SASSY by (Hybinette et al., 2006) is a scalable agent based simulation system that sits as a middle-ware between an agent-based API and a Parallel Discrete Event simulation (PDES) kernel. The difference in SASSY compared to (Logan and Theodoropoulos, 2001) and (Riley and Riley, 2003) is that the ABM framework can be built up from existing standard PDES kernels. (D’Souza et al., 2007) proposes an innovative method of executing mega-scale Agent-Based Models in the massively parallel Graphics Processing Unit (GPU). Although, it is well established that this method can lead up to considerable speed gains, we feel that the expressiveness of Agent-based models that can be run on this platform is restricted. A similar framework is Flame GPU, built on-top the FLAME ABM framework (Kiran et al., 2010), and has successfully been applied on project EURACE to simulate the European economy model (Deissenberg et al., 2008).

Concerning Haskell, our solution is the first Logo-based modeling framework to be implemented in the Haskell programming language. There have been, however, other Haskell simulation packages. E.g. Aivika is a promising Haskell library that provides extensive system dynamics and discrete event simulation. Event-monad, as the name suggests, provides an event monad and monad transformer; it can be used as a low-level helper library to build a simulation framework. Users can create an event-graph simulation system and schedule events to it. In principle, it does not employ any parallelism, but it could theoretically be used together with some parallel strategy to exploit parallelism. Hasim is a library for process-based Discrete Event Simulation in Haskell. It does not employ any kind of parallelism.
6 CONCLUSION AND FUTURE WORK

We have presented HLogo, a variant of the NetLogo framework that utilizes Software Transactional Memory to benefit in parallelism. Although the agent execution may become non-reproducible, we still believe that there is room for applying HLogo, considering the potential speedup and the fact that NetLogo is widely used as well for constructing Multi-Agent Systems (where reproducibility is not a factor). The HLogo language is embedded as a Domain-Specific Language in Haskell, which has the advantage of inheriting the Haskell’s module system and allowing the ease of import of existing Haskell libraries. Furthermore, the DSL is typed for all its expressions and agent commands, which adds a certain level of safety to Logo models. We managed to port NetLogo models to HLogo and benchmark two of those against the frameworks. Execution results show that HLogo is faster than NetLogo when increasing the number of cores.

One of our future plans regarding the extension of the presented approach, is to investigate if the atomic construct can be automatically inserted in certain places of HLogo programs that will retain consistency while in the same time admit reproducibility; possibly as the result of static program analysis. Moreover, some STM transactions can be accelerated after applying certain optimizations, e.g. when a cow moves, it may wiggle atomic (do rt=<<random 50; lt=<<random 50). The above code can be optimized by atomic (do i<-random 50; j<-random 50; lt (j-i)) which is faster since the STM transaction log is shortened through combining two modifications of the turtle’s heading (right, left) to one (left). The program remains consistent since this code runs atomically: no other agent could have, in any way, witnessed the intermediate modification.

Concerning HLogo’s engine, the ‘divide and conquer’ method of splitting the workload to the threads does not involve any intelligence; it simply slices equally the given agentset. This naive method can possibly lead to unnecessary STM conflicts. By exploiting the spatial characteristics of the model we could better (more cleverly) assign the work to the threads, so that it minimizes the number of conflicts (retries). In other words, interdependent agents should end up in the same thread, whereas independent agents should be distributed to different threads (and CPU cores). This work clustering could be static (on initialize), or adaptive (during runtime execution).

Since the Cloud has become very accessible, we would like to investigate the execution of HLogo simulations in a distributed setting. Compared to a single physical multi-core system, a distributed setting can enable Agent Based Models to run in High-performance Computing (HPC); there is also the extreme case where the model cannot fit in a single shared memory and has to be distributed to multiple processing nodes. Haskell technologies, such as Distributed Software Transactional Memory (Kupke, 2010) or Cloud Haskell (Epstein et al., 2011) could help us achieve this task.

Finally, we have to note that our ideas are in no sense restricted to NetLogo-like simulations; our approach is framework-agnostic and thus could be easily applied to other ABM frameworks too. However, since NetLogo is a well-established platform, with a large user base, we would also like to experimentally extend the existing NetLogo engine with support for one of the many Scala STM implementations. We argue that the NetLogo language again needs only to be extended with an atomic construct, thus, remaining close to NetLogo’s syntax and utilizing the original Graphical User Interface.

ACKNOWLEDGEMENTS

This work was funded partially by the EU project FP7-610582 ENVISAGE: Engineering Virtualized Services (http://envisage-project.eu). All the benchmarks in this work were carried out on the Dutch national HPC e-infrastructure, kindly provided by the SURF Foundation (http://surf.nl).

REFERENCES


