Turtle Ballet: Simulating Parallel Turtles in a Nonparallel LOGO Version

Erich Neuwirth

University of Vienna, Dept. of Statistics and Decision Support Systems
Computer Supported Didactics Working Group
A 1010, Universitaetsstr. 5/9 Vienna, Austria
Tel: + (43 1) 4277 38624; Fax: + (43 1) 4277 9386
erich.neuwirth@univie.ac.at

Abstract

StarLogo and NetLogo implement multiple parallel turtles with powerful abstraction mechanisms for multi agent parallelism. We will show how to implement similar concepts in a more standard version of Logo, namely MSWLOGO. This also demonstrates that we can extend the fundamental concepts of LOGO relatively easily. We also show how powerful high level abstractions in LOGO can support extending the language.

Keywords: Parallel programming, graphics programming, advanced Logo programming

1. Introduction

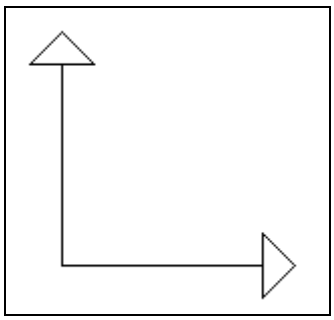
StarLogo and NetLogo implement grid based turtle worlds with multiple turtles performing actions in parallel. Compared with "classic" Logo, this paradigm allows for a completely new set of activities. Basically, the turtles live on a grid, and they can manipulate the state (e.g. color) of the grid cells. The grid cells, however, are the smallest "colorable" units. So curve drawing by turtle movement, one of the original Logo concepts, cannot be done in these new variants of Logo.
Therefore we decided that it would be worthwhile to take a more standard version of Logo, namely MSWLogo, and add parallel turtles to it. These turtles will still be "curve drawing turtles", and using our new toolkit we will be able to study concepts like envelopes of curve sets in a way not easily done without parallel "curve" turtles.
It is not easy to give convincing graphical examples of the new facilities in print since parallelism has to be "watched while developing", and in print we only could give snapshots which could also be created in a nonparallel way. Therefore, we will discuss our toolkit and illustrate the ideas and possibilities while we are developing the tools.

2. Basic design

In our project, we are using MSWLogo. This version of Logo has simple support for multiple turtles. It has 1024 turtles (numbered from 0 to 1023). These turtles may have different positions, and they can be moved independently. **`setturtle`** is the command for indicating the turtle to be addressed, so a prototypical code segment using this command might look like this:

```
setturtle 1
forward 100
setturtle 2
right 90
forward 100
```

and will produce a screen image like this:



The Logo function **turtle** gives the number of the last "addressed" turtle.
This mechanism has the disadvantage that when dealing with many turtles simultaneously we need a lot of bookkeeping about the current turtle.
To make programming clearer we define a new procedure, **tell**. Its behavior is described by the following example:

```
tell 1 [forward 100]
```

will move turtle number 1 100 steps forward.
It is easily implemented by

```
tell :who.single :action
localmake "oldturtle turtle
setturtle :who.single
run :action
setturtle :oldturtle
end
```

We will immediately refine this procedure. The purpose of the refinement is that it also accepts commands like

```
tell.turtle 20 [forward 10 right 90 forward 10]
tell.turtle 20 [[forward 10] [right 90] [forward 10]]
```

This can be achieved by the following definition:

```
to tell.turtle :who.single :action
if not listp first :action ~
       [tell.turtle :who.single (list :action) stop]
localmake "oldturtle turtle
setturtle :who.single
foreach :action [run ?]
setturtle :oldturtle
end
```

This definition uses two very powerful Logo commands, **foreach** and **run**.
run takes a list as argument and executes this list as a "small Logo program".
foreach takes a list as a first argument and a template (a program segment with a placeholder) as its second argument and replaces the placeholder (in our case **?**) with the elements of the list, and then executes this program segment for each list element. **run** is part of most Logo implementations, **foreach** is a

predefined library procedure in MSWLogo, but it can be implemented in any Logo version containing **run**. BrianHarvey's book gives a good introduction on how to implement **foreach** and many other high level functions and procedures.
Using this definition we can define a new command:

```
to tell.turtles :who.crowd :action
foreach :who.crowd [tell.turtle ? :action]
end
```

From now on we will use the function **iseq** quite often. **iseq** produces a list of consecutive integers, e.g. **iseq 1 4** outputs **[1 2 3 4]**.
Additionally, we need to remember that the function **turtle** always returns the number of the current turtle.

tell.turtles is best illustrated by the following example:

```
tell.turtles iseq 1 4 [[forward 10 * turtle] ~
       [right 90] [wait 100] ~
       [forward 10 * turtle] [right 90] [wait 100]]
```

The list of turtle commands is executed for each of the turtles given in the list **iseq 1 4**, and the list is completed for one turtle before it is executed for the next turtle. This in not what we really want, we want all our turtles to execute [forward 10 * turtle], and then we want them to execute [right 90] and so on. We can describe this in the following way: each element of our command list should be executed by **tell.turtles** for all turtles before the next element of the command list is being executed. We will define a new procedure for accomplishing this goal. This procedure will be called **do**.

```
to do :who :action
foreach :action [tell.turtles :who ?]
end
```

This procedure does what it is supposed to do, but we still cannot execute loops in parallel. Let us describe a prototype of what we want:

```
repeat 360 [forward 1 right 1]
```

is the standard Logo way or drawing a circle
Running the program

```
tell.turtles iseq 1 10 [setx 10 * turtle]
do iseq 1 10 [repeat 360 [forward 1 right 1]]
```

lets turtle 1 complete its circle, then turtle 2 complete its circle and so on. We want the turtles to draw their circles "in sync". So we want a parallel version of repeat. We could implement **repeat.par**

```
to repeat.par :who :n.of.repeats :action
repeat :n.of.repeats [tell.turtles :who :action]
end
```

Running

```
tell.turtles iseq 1 10 [setx 10 * turtle]
repeat.par iseq 1 10 360 [[forward 1][right 1]]
```

illustrates why we are talking about "turtle ballet". Since all the turtles leave traces, we even could talk about "turtle ice-skating ballet".
Alas, this implementation has some disadvantages.
When we have a program

```
setx 10 * turtle
repeat 360 [forward 1 right 1]
```

in theory we could turn it into a parallel program by the following modification:

```
do iseq 1 10 [setx 10 * turtle]
do iseq 1 10 [repeat 360 [forward 1 right 1]]
```

This will not, however, produce a “really” parallel program because the circles will not be drawn “in sync”. Instead of using **repeat.par** we can solve this problem (at least for now) by redefining **tell.turtles**.

```
to tell.turtles :who :action
if (first :action) = "repeat ~
        [repeat (invoke butlast butfirst :action) ~
            [foreach :who [tell.turtle ? last :action]] stop]
foreach :who [tell.turtle ? :action]
end
```

Here we use another high level Logo function, **invoke**. The idea is that a complete **repeat** instruction has the following structure:
repeat *repeatcount* [instructionlist]
In simple cases *repeatcount* is a number, but it also might be an expression producing the value needed. Invoke takes this expression just before the instruction list and evaluates it.
This is the general purpose of **invoke**: it takes a list which “really” is a Logo expression and returns the value produced by this expression.
With this definition, the example above will work as intended; the turtles will draw circles in a synchronized way. The following example will work as intended

```
do iseq 1 10 [[setx 10 * turtle] [repeat 360 [fd 1 rt 1]]]
```

and the next example will produce Brownian motion (i.e. random movements of the turtles).

```
do iseq 1 100 [[hideturtle] ~
       [repeat 100 [fd 5 right random 360]]]
```

To avoid misconceptions it is important to note that the repeat count (100 in the example above) may not be different for different turtles, trying, for example

```
do iseq 1 2 [repeat 10 * turtle [forward 10 right 10]]
```

will not work.
In the next step, we want to modify the random movement program slightly: we want the turtles to stop when they are more than 100 steps away from the center. Standard Logo has a built-in function **distance**, and for a “one turtle Logo” **distance [0 0]** gives the distance of the turtle from the center of the screen. There are more functions related to the position of the turtle, e.g. **pos**, **xcor**, and **ycor**. Since we are dealing with multiple turtles here, we have to make clear which turtle we are addressing, and this is done by a new function:

```
to ask.turtle :who :question
localmake "oldturtle turtle
setturtle :who
localmake "result (invoke :question)
setturtle :oldturtle
output :result
end
```

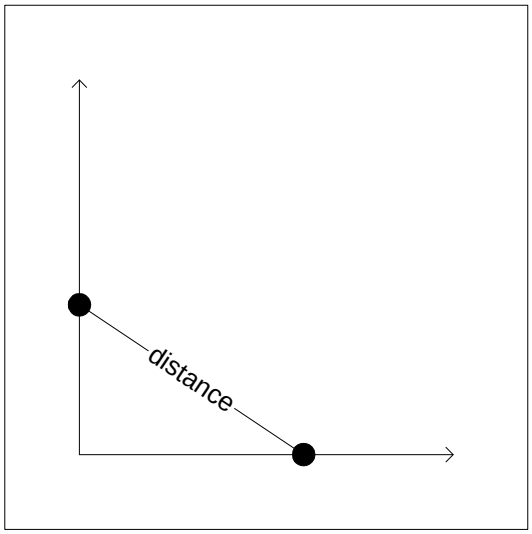
ask.turtle 3 [pos] will return the position of turtle 3.
We will also ask all the turtles some questions at the same time, and this is implemented by the following function:

```
to ask.turtles :who.crowd :question
output map [ask.turtle ? :question] :who.crowd
end
```

map is a high level Logo function that evaluates an expression for a list of inputs and returns the corresponding list of outputs.
Using **ask.turtle** we now can modify our Brownian motion code:

```
do iseq 1 100 ~
  [[ht] ~
   [repeat 100 [if (ask turtle [distance [0 0]]) < 100 ~
         [forward 10 right random 360]]]]
```

This little program will stop all the turtles when they are "too far away" from the origin.
tell.turtle, **tell.turtles**, **do**, **ask.turtle**, and **ask.turtles** now already implement a simple "synchronized turtles" microworld.
Let us use our microworld for a mathematical example, an envelope curve.
We start with a given distance and put a turtle on the x-axis. Then we move the turtle from the point on the x-axis to the point on the y-axis that has the given distance from the original point on the x-axis.



We will use many turtles along the given distance on the x-axis, and have them all travel the same distance to reach the y-axis.
So we need the following function:

```
to ytarget :xorigin :dist
output sqrt :dist * :dist - :xorigin * :xorigin
end
```

Now we already can arrange our turtles:

```
to setup.envelope :n.of.turtles :dist
tell.turtles iseq 0 :n.of.turtles – 1 ~
      [setx :distance * turtle /  (:n.of turtles – 1)]
tell.turtles iseq 0 :n.of.turtles – 1 ~
      [setheading towards list 0 ask.turtle turtle [xcor]]
end
```

and we can let them run:

```
to envelope :n.of.turtles :dist
setup.envelope :n.of.turtles :dist
tell.turtles iseq 0 :n.of.turtles – 1 [forward :dist]
end
```

This procedure does not yet produce the animation effect we would like to see; each turtle travels its complete path before the next turtle starts. We can achieve "synchronized" movement by some simple modifications (we also have to hide the turtles to see the evolving pattern more clearly):

```
to setup.envelope :n.of.turtles :dist
tell.turtles iseq 0 :n.of.turtles – 1 ~
      [setx :distance * turtle /  (:n.of turtles – 1)]
tell.turtles iseq 0 :n.of.turtles – 1 ~
      [setheading towards list 0 ask.turtle turtle [xcor]]
tell.turtles iseq 0 :n.of.turtles – 1 [hideturtle]
end

to envelope :n.of.turtles :dist
setup.envelope :n.of.turtles :dist
do iseq 0 :n.of.turtles – 1 [repeat :dist [forward 1]]
end
```

Here is another example: We put turtles equally spaced along the x-axis, and then we let them draw circles, but with a twist: each turtle invisibly travels a segment of its circle before all the turtles start simultaneously. The effect is rather hard to describe in words and in pictures,
It heavily relies on the dynamic visualization, so the next procedure really has to be run to see the effect.

```
to circles
tell.turtles iseq 0 100 [hideturtle]
tell.turtles iseq 0 100 [setx 3.6 * turtle]
tell.turtles iseq 0 100 [penup]
tell.turtles iseq 0 100 [repeat int 3.6 * turtle ~
     [forward 1 right 1]]
tell.turtles iseq 0 100 [pendown]
tell.turtles iseq 0 100 [forward 1 right 1]
do iseq 10 100 [repeat 360 [forward 1 right 1]]
end
```

3. Extending the turtle ballet microworld

The basic implementation we discussed in the previous section allows synchronized turtle movement. The general pattern for programs usually is:
Use **`tell.turtles`** to set up starting positions and headings, then use **`do`** to perform parallel turtle movements. Here is a minimal example of this kind of program:

```
to pattern
tell.turtles iseq 0 99 [setheading 3.6 * turtle]
tell.turtles iseq 0 99 [hideturtle]
do iseq 0 99 [repeat 360 [forward 1 right 1]]
end
```

When programs get more complex, we want to be able to define subroutines. Let us consider a simple case:

```
to do.something
repeat 100 [forward 1 wait 1]
right 90
repeat 100 [forward 1 wait 1]
left 90
repeat 100 [forward 1 wait 1]
right 90
repeat 100 [forward 1 wait 1]
left 90
end
```

Running do **iseq 1 4 [[setheading 90 * turtle] [do.something]]** does not do what we would like it to do. It does not perform **do.something** in parallel. A parallel version of **do.something** would have to look like this:

```
to do.something.par
do iseq 1 4 [repeat 100 [forward 1 wait 1]]
do iseq 1 4 [right 90
do iseq 1 4 [repeat 100 [forward 1 wait 1]]
do iseq 1 4 [left 90
do iseq 1 4 [repeat 100 [forward 1 wait 1]]
do iseq 1 4 [right 90
do iseq 1 4 [repeat 100 [forward 1 wait 1]]
do iseq 1 4 [left 90]
end
```

Simply stated, each line of **do.something.par** is produced by putting a line of **do.something** in brackets and prefixing it with **do iseq** and the relevant inputs to **iseq**. This can be automated. A simplified version of a function parallelizing a procedure could look like this:

```
to parallel.version :procname
output fput first text :procname ~
     map [list "do ":who ?] butfirst :text :procname
end
```

Here we use the function **text**, which returns the definition of a procedure (or a function) as a list which can be manipulated. The dual procedure is **define;** it takes a list that represents a procedure and defines a procedure accordingly. Using **text** we can define

```
to parallelize :procname
define word :procname ".par parallel.version :procname
end
```

parallelize "do.something then defines a new procedure,
do.something.par, and the new procedure has a parameter **:who** which accepts the turtle set for which the parallelized version of the procedure should be run.
Of course, a user-defined procedure can call other user-defined procedures, and therefore the full version of should take care of the fact that procedures called from a parallelized procedure should also be parallelized. This can be accomplished by „interweaving" **do** and **parallelize**.
There is one more problem which we have not considered yet. Procedures can contain the command **stop**. Normally, **stop** stops the execution of a procedure completely. In a parallel procedure, **stop** should not do this, it just should suspend execution of all "future" commands in a procedure for the turtle for which the stop condition was fulfilled. This can be implemented by keeping a list of currently active turtles and removing turtles from this list when they meet a stop condition. Giving the full code for this additional facility would lengthen this paper too much. Fur further investigations, an enhanced version of the code from this paper is available on the WWW from
http://mailbox.univie.ac.at/erich.neuwirth/multiturtle/ .

To be able to implement this changed behavior of **`stop`**, we need one more of the advanced capabilities of Logo, namely redefining primitive operations. The implementation of our "stop turtles separately" idea is accomplished by using this facility.
MSWLogo does not keep a "memory" of pencolor for multiple turtles. To make multicolored drawings with multiple turtles, we also need to redefine **`pencolor`** and **`setpencolor`**. This is also implemented in the code available from the WWW.

4. Concluding remarks

StarLogo and NetLogo have somewhat changed the "feel" of typical Logo projects. Curve drawing especially has been almost abandoned. From a didactical point of view, combining curve drawing with parallel turtles offers some very nice possibilities for investigation of "classical mathematical" interest, and so brings together some of the original concepts underlying Logo with the modern facilities of parallel programming.
The toolkit described in this paper can be used to investigate these curve sets.
Additionally, our code can serve as a prototypical example for using some of the more powerful operators of Logo (**`run`**, **`invoke`**, **`define`**, **`text`**…) to extend the language itself. Adding pseudo parallel execution to an inherently nonparallel language seems like a major undertaking. Demonstrating that this can be achieved in Logo with relatively little code (the relevant code is less than 200 lines in the code available from the WWW) shows that Logo is an extremely powerful and extensible language. Additionally, the code is more or less readable and thereby demonstrates that implementing new ideas is achievable with a sound educational tool in a transparent way.

References

StarLogo Pages at MIT, http://el.www.media.mit.edu/groups/el/Projects/starlogo/
StarLogo Pages at Northwestern University, http://ccl.northwestern.edu/cm/starlogoT/
NETLogo Pages, http://ccl.northwestern.edu/netlogo/
MSWLogo, http://www.softronix.com/logo.html
Multiturtle Logo code by Erich Neuwirth, http://sunsite.univie.ac.at/multiturtle/
Brian Harvey, Computer Science Logo Style, MIT Press 2000.