Simulating Cellular Automata: A Challenge for Programming Languages

alphabetically
Richard J. Fateman
Computer Science Division
University of California
Berkeley, CA. 94720, USA
`fateman@cs.berkeley.edu`
and
.. deleted by request ..

sometime in 1994, retypeset Jan, 2011

Abstract

There is a long history of computer simulation of various natural and artificial phenomena. Representations based on cellular automata are attractive because they tend to be conceptually simple to set up, and in a suitable framework can be observed "at work." Writing efficient and easily read/modified programs to run such simulations is a challenge to the designers and implementors of a programming language. More specifically, the design task must solve the problem of making the programs easy to write and read. The implementation task is providing a path to sufficiently efficient code so that running sufficiently many simulation step for the purpose at hand steps will be feasible with available computing resources.

Here we compare a few languages on a particular challenge problem. [2011 update: this is an old paper that sat unchanged and unpublished for years because several co-authors did not want their names to be associated with their programs or this paper. Presumably ALL the programs would run much faster on up-to-date software and hardware, and even the relative timing comparisons would need to be recomputed.]

1 The Challenge

This problem was posed as a challenge to advocates of various programming languages on a netnews (sci.math.symbolic discussion in 1994. [1], [2].

The basic idea is to simulate a forest-fire and subsequent evolution in a one-dimensional line of n trees and spaces between trees. At time t_0 an initial arrangement of trees is set up by placing a tree at each of the n positions with independent probability p. For succeeding generations t_i for $i > 0$ we perform three steps.

- Ignite each tree with probability f;
- For every tree adjacent to a tree that is on fire, we instantaneously ignite it and all its neighbors. The spread of the fire is inhibited by an empty space. When all the trees are "finished burning" we then replant:
- In each place that a tree does not exist (was just burned or was empty at ignition) plant a tree with probability p.

The end-conditions are set for tree 0 to be adjacent to tree $n - 1$ so that the fire can propagate in a loop. One might envision the trees as being arranged single-file around the coast of an island whose interior is a desert.

Why is this interesting as a simulation? One can consider various settings of the parameters and see how the density of trees varies. For example, if the probability of a fire is 0, but the probability of a sprout is positive, the forest fills up. A small probability of a fire can completely wipe out a dense forest. Slower growth and/or higher fire frequency may maintain a more balanced simulated "ecosystem." One can change the rules. For example, have some fire-breaks that are permanently fire-proof, and see what happens.

2 How to Represent the Data

It is natural to abstract this forest into a one dimensional array or list of n objects each of which indicates the presence or absence of a tree, and (depending upon programming) may also have to represent a tree on fire. For purposes of fire-propagation, one needs to be able to determine, for each tree ignited, whether it has a tree at its left or right.

Since the evolution of a forest over time is of interest, the data structure should be mapped into a graphical display. For example one could animate the forest, or have a 2-dimensional display where adjacent "scan lines" represent the evolving sequence of tree/space generations.

Other representations are possible. If the trees are expected to be quite rare, one could use a set of those locations at which trees are present. If spaces are rare, those could be stored instead.

In writing programs for this simulation, two other ideas were examined by one or more of the programmer/authors.

- The boundary conditions are most likely imposed by an implementation of modular arithmetic where $(n - 1) + 1 = 0$ rather than n. If modular arithmetic is slow, it may pay to find an alternative.
- This simulation requires a potentially large number of random numbers to implement the probabilistic rules for sprouting and igniting trees. Simulations based on the use of a pseudo-random number generator algorithm are quite common: there are many algorithms; some of them "not very random" and some "not very fast." Some common and notorious pseudo-random number generators are disqualified on both counts! A good fast algorithm will improve the speed, and perhaps the "quality" of the simulation.

2.1 The Mathematica programs

Richard Gaylord initially posted the following version.

```
The steps of the algorithm and the code to implement them will be given
together.

(1) A forest preserve of length n, consisting of empty sites and tree
sites is created using

forestPreserve = Table[Random[Integer], {n}]

All of the sites in the forest preserve are updated in each time step,
according to the following sequence of steps:

(2a) Trees catch fire with probability f and empty sites sprout trees with
probability p, using a set of transformation rules to turn 0's into 1's
with probability p, and 1's into 2's with probability, f
```

```
treeGrowIgnite =
forestPreserve /. {0 :> Floor[1 + p - Random[]], 1 :> Floor[2 + f -
Random[]]}
```

(2b) All tree sites adjacent to an ignited tree site (ie., trees in the same forest) ignite. This accomplished using the repeated application of a set of transformation rules

```
forestIgnite =
     treeGrowIgnite //. {{a___, 2, 1, b___} -> {a, 2, 2, b},
                         {a___, 1, 2, b___} -> {a, 2, 2, b},
                         {2, c___, 1} -> {2, c, 2},
                         {1, c___, 2} -> {2, c, 2}}
```

(note: The first two transformation rules change sequences in the list having the form .., 1, 2,.. and .., 2, 1, .. to .., 2, 2, .. . and the last two transformation rules implement periodic boundary conditions by treating trees at each end of the system as belonging to the same forest.)

(2c) Ignited trees burn down using

```
forestNew = forestIgnite /. 2 -> 0
```

The sequence of steps 2a-c can be combined in an anonymous function which can be applied to any forest preserve configuration

```
pyro =
  (treeGrowIgnite = # /. {0 :> Floor[1 + p - Random[]], 1 :> Floor[2 + f -
Random[]]};
   forestIgnite = TreeGrowIgnite //. {{2, c___, 1} -> {2, c, 2},
                                      {1, c___, 2} -> {2, c, 2},
                                      {a___, 2, 1, b___} -> {a, 2, 2, b},
                                      {a___, 1, 2, b___} -> {a, 2, 2, b}};
   forestNew = forestIgnite /. 2 -> 0)&
```

where # represents the forest preserve configuration.

(3) Step 2 is repeated m times using

```
NestList[pyro, forestPreserve, m]
```

These pieces of code can be combined into a program.

Program

```
smokeyTheBear[n_, p_, f_, m_] :=
   Module[{},
       forestPreserve = Table[Random[Integer], {n}];
       pyro =
          (treeGrowIgnite = #/.{0 :> Floor[1 + p - Random[]], 1 :> Floor[2
+ f - Random[]]};
```

```
           forestIgnite = TreeGrowIgnite //. {{2, c___, 1} -> {2, c, 2},
                                              {1, c___, 2} -> {2, c, 2},
                                              {a___, 2, 1, b___} -> {a, 2,
2, b},
                                              {a___, 1, 2, b___} -> {a, 2,
2, b}};
           forestNew = forestIgnite /. 2 -> 0)&;
       trees = NestList[pyro, forestPreserve, m]
        ]
```

2.2 The Lisp programs

Richard Fateman contributes the Lisp program discussed below.

This program has gone through quite a few revisions, first to get it correct (the propagation rules for the fire were not so clearly stated as we have given them above) and then to make it faster.

We considered the representation of a forest as a list, vector, bitmap, and byte-array. Using a profiling tool, we found that the vast majority of the time was actually spent in the Lisp system's high-quality but overly-general built-in random number generator. We were able to try several better ones. The first was written in Lisp; another faster one was used from the UNIX operating system math library on our system (`accrans`). Then because of a technicality[1] we changed the program to use integers between 0 and 2^{24} instead of floats between 0.0 and 1.0. Avoiding floating-point numbers entirely was a good idea.

Since random-number generation took 95% of the time initially, changes to this component to make it faster improved the overall time by about a factor of 20.

Observing that the function `mod` was not compiled into neat code, we replaced `(setf k (mod (- k 1) n))` by `(setf k (if (= k 0) n-1 (- k 1)))` eliminating all divisions. Similarly, `abs`, the absolute value was overly general, so we replaced it, too.

Rather than lead the reader through one level of elaboration after another, we present the code, with apologies stated in the comments (all text to the right of a semicolon (;) is a comment) among the programs, as finally run on an HP computer system.

```
;;;-*- mode: Common-Lisp; -*-

;;;
;;; forest fire simulation, fast UNIX System V version (on HP Allegro 4.2)
;;; uses lrand48 congruential integer random number generator.
;;; uses byte arrays

(declaim (optimize (speed 3)(safety 0))) ;; compile for speed, not safety

;;; set up random number generator
;;; mumbo jumbo for the next 20 lines, most of which are comments.

;; set up the entry point to the library "libm" library routine lrand48

(unless (fboundp 'lrand48) ; read this in, only if not yet loaded
  (load "" :unreferenced-lib-names '("_lrand48")
:system-libraries '("m"))
;; (lrand48) as used here produces integers |j| <2^29-1, even though
;; the unix manual says slightly otherwise. Lisp fixnums lose the high
;; bit.
```

[1]The cost of using "unboxed" floating-point numbers in Lisp

```
;; attach the "foreign function" entry point to a lisp function
;; this defines lrand48 as a lisp function that returns a random
;; fixnum in (-2^29, 2^29).

  (ff:defforeign 'lrand48    :arguments '() :entry-point "_lrand48"
      :call-direct t :callback nil :arg-checking nil
      :return-type :fixnum))

;; We discovered that the generic abs function was not compiled in-line.
;; (We view this as a compiler defect that should be changed), but
;; in the tradition of using Lisp as an assembly-level language,
;; here's a way to fix this code. Use iabs instead of abs, and
;; define iabs as:

(defmacro iabs(f)
  "in-line fixnum absolute value"
  (let ((x (gensym)))
    `(the fixnum
  (let ((,x ,f))
    (declare (fixnum ,x))
    (if (< ,x 0) (- (the fixnum ,x)) ,x)))))

;;;; Finally we can write the simulator

;; initialize forest  1-> tree, 0->empty

(defun init-forest (size)
  (let ((oneline  (make-array size :element-type '(unsigned-byte 8))))
    ;; initialize forest  1-> tree, 0->empty
      (declare (fixnum size)
               (type (simple-array (unsigned-byte 8) (*)) oneline))
    (dotimes (i size)
      (setf (aref oneline i) (mod (the fixnum (lrand48)) 2)))
    oneline))

(defun TreeGrowIgnite (forest p229 f229 size)
    (declare (fixnum size p229 f229)
     (type (simple-array (unsigned-byte 8) (*)) forest))
    (let ((r 0))
      (declare (fixnum r))
      (do ((i 0 (1+ i)))
((= i size) forest)
(declare (fixnum i))
(setf r (aref forest i))

(cond ((= r 0)
       (if (< (iabs(lrand48)) p229) (setf (aref forest i) 1)))
      (t
       (if (< (iabs(lrand48)) f229) (setf (aref forest i) 2)))))))

;; Spreadfire is clumsy -- how to make it neat? I think that
;; this is not really a traditional CA "generation" maker because
;; it is an O(size) operation to go from one generation to the next.
;; (that is, a generation (2 1 1 1 1 ...) or (1 1 1 ... 2)
;; needs to propagate across size-1 cells.)

(defun spreadfire(forest size)
  (declare (fixnum size)
   (type (simple-array (unsigned-byte 8) (*)) forest) )
  (let((h (1- size))
       (sizem1 (1- size)))
    (declare (fixnum h sizem1))
    (loop (if (< (the fixnum h) 0)
      (return forest));; look through the forest
      ;; if we find a tree on fire,
  (cond ((= (aref forest h) 2)
 (setf (aref forest h) 0);burn it down
 ;; spread to adjacent areas
 (spreadright h sizem1 forest)
 (setf h (min h (the fixnum (spreadleft h sizem1 forest)))))
(t (decf h))))))

;; Two subsidiary functions to spreadfire.  We initially defined
;; these INSIDE spreadfire via a "labels" construct -- a more
;; traditional structured form that is also more typical of Scheme
;; programs,  but that seems to produce slower code in dealing with
;; lexical variables.  So in the more ‘‘old-fashioned’’ Lisp form, we
;; use separately compiled functions to spread the fire right and left.

(defun spreadright (k sizem1 forest)
;;k is starting position, initially had a 2
  (declare (fixnum k sizem1)
   (type (simple-array (unsigned-byte 8) (*)) forest))
  (do* ((j   (if (= k sizem1) 0 (1+ k));(mod (1+ k) size)
     (if (= j sizem1) 0 (1+ j))))
      ;; test termination condition
      ((= (aref forest j) 0) ;firestopper
       j)
    (declare (fixnum j))
    (setf (aref forest j) 0)))

(defun spreadleft (k sizem1 forest)
 ;; almost the same as spreadright
  (declare (fixnum k)
   (type (simple-array (unsigned-byte 8) (*)) forest))
  (do* ((j (if (= 0 k) sizem1 (1- k));(mod (1- k) size)
   (if (= 0 j) sizem1 (1- j))))
      ;; test termination condition
      ((= (aref forest j) 0) ;firestopper
       j)
    (declare (fixnum j))
    (setf (aref forest j) 0)))
```

```
;; Because floating point comparisons require carrying along
;; "pointers to floats" it pays to convert all the numbers and
;; comparisons to single-word "fixnum" integers, which in this
;; lisp are of magnitude less than 2^29.
;; The following top-level functions make this transformation:
;; multiply the probability, expected to be in the interval [0,1]
;; by (2^29)-1, and convert to an integer.  These numbers then
;; are used in conjunction with an integer pseudo-random number
;; generator to make decisions in the model.

(defun smokeytty (size p f m) ;; debuggging version - use printing display
  (let* ((oneline (init-forest size))
 (p (truncate (* p #.(1- (expt 2 29))))) ;; fixnum arith is faster..
 (f (truncate (* f #.(1- (expt 2 29))))))
    (declare (fixnum p f m size))
    (dotimes (i m)
      (treegrowignite oneline p f size) ;randomly grow/light
      (spreadfire oneline size) ; spread the fire
      (dotimes (i size)
        ;; print a ``+'' for a tree, a space for no tree
(format t "~a" (if (= 1 (aref oneline i))  "+" " ")))
       ;; take a new line for each generation
      (format t "~%"))))

(defun quietsmokey (size op of m) ;; no-printing version -- for timing
  (let*  ((oneline (init-forest size))
 (p (truncate (* op #.(1- (expt 2 29)))))
 (f (truncate (* of #.(1- (expt 2 29))))))
    (declare (fixnum p f m size))
    (dotimes (i m)
      (treegrowignite oneline p f size) ;randomly grow/light
      (spreadfire oneline size) ; spread the fire
)))

;;; set up for common-windows, an interface to the X window system

(eval-when (compile load)(use-package :common-windows))
(eval-when (load eval)
 (unless (common-windows-initialized-p)
   (setf *window-manager-titles-p* t)
   (initialize-common-windows) ))

(defun smokey (size op of m) ;; display version
  ;; p is prob of a tree growing in an empty spot  0<= p <=1
  ;; f is prob of a tree igniting 0<=f<=1
  (let* ((sherwood
          (make-window-stream :left  200  :bottom 100
      :width size :height m
      :title "forest"
      :bits-per-pixel 1
      :activate-p t))
 (p (truncate (* op #.(1- (expt 2 29)))))
 (f (truncate (* of #.(1- (expt 2 29)))))
 (oneline (init-forest size))
 (bm (make-bitmap :width size :height 1 :bits-per-pixel 1)))
    (declare (type (simple-array (unsigned-byte 8) (*)) oneline)
     (fixnum p f size m))
    (dotimes (i m)
      (treegrowignite oneline p f size) ;randomly grow/light
      (spreadfire oneline size) ; spread the fire
    ;;deposit the bits into a bitmap bm, and then put them into
    ;;the forest.
      (dotimes (k size)
(declare (fixnum k))
(setf (bitmap-bit bm k 0) (aref oneline k)))
      (bitblt bm 0 0 sherwood 0 i)) ;put each line in the display
    sherwood))
```

It turns out that an alternative model, where only a set number of pseudo-random numbers in $[0, 2^{29})$ are generated, and then re-used, is even faster. One version of the code for this looks like:

```
(let ((size 0) ;;random number encapsulation
      (r (make-array 1 :element-type 'fixnum)) ;for declaration consistency
      (ptr 0))

  (declare (fixnum s ptr size)
   (type (simple-array fixnum (*)) r))

  (defun init-lrand48 (s)
    (declare (fixnum s))
    (setf r (make-array s :element-type 'fixnum))
    (dotimes (i s)
      (declare (fixnum i))
      (setf (aref r i) (iabs(lrand48))))
    (setf size s)
    (setf ptr (1- s))
    r)

  (defun next-lrand48() ;; decrement the pointer, return the entry in r
    (aref r (if (= ptr 0) (setf ptr (1- size))(decf ptr)))))
```

Now instead of using `(iabs(lrand48))`, one uses `(next-lrand48)`. This requires first initializing the pseudo-random number table by saying, for example, `(init-lrand48 10003)` when the program is loaded in. It saves run-time by replacing a call to generate a pseudo-random number, to an array reference and a counter increment.

If we were to try to make the program prettier, we would go back and replace the constants 0, 1, 2 with names, such as `no_tree`, `tree`, `burning_tree`. We would also change the test names to `s_burning` etc. This could have been done in the Mathematica code as well, but in such a small piece of code, the extra level of abstraction seems dubious.

2.3 The SML programs

SECTION DELETED

2.4 The MAGMA program

SECTION DELETED by request of author.
[note added in 2011. See the example program in
http://magma.maths.usyd.edu.au/magma/pdf/examples.pdf
]

2.5 Improved Mathematica code

AUTHOR deleted

```
MySmokey[n_Integer, p_, f_, m_Integer] :=
  With[{fcomp = Compile[{rand},Evaluate[Floor[2+f-rand]]],
    pcomp = Compile[{rand},Evaluate[Floor[1+p-rand]]]},
    Module[{forestPreserve,forestIgnite,spreadFire,treeGrowIgnite},

      forestPreserve = Table[Random[Integer],{n}];

      SetAttributes[treeGrowIgnite,Listable];
      treeGrowIgnite[0]:= pcomp[Random[]];
      treeGrowIgnite[1]:= fcomp[Random[]];

      forestIgnite[{path___?(#===1&),2, rem___, 1}]:=
        {spreadFire[path,2],spreadFire[rem,2]} /. 2->0;
      forestIgnite[{1, rem___, 2,path___?(#===1&)}]:=
        {spreadFire[2,rem],spreadFire[2,path]} /. 2->0;
      forestIgnite[{rem__}]:= {spreadFire[rem]} /. 2->0;
      spreadFire[l___,2,1,r___]:=
        Sequence[spreadFire[l,2],spreadFire[2,r]];
      spreadFire[l___,1,2,r___]:=
        Sequence[spreadFire[l,2],spreadFire[2,r]];
      spreadFire[rem__]:= rem; (* Stopping criterion. *)

      NestList[ forestIgnite[treeGrowIgnite[#]]&, forestPreserve, m]
    ]
  ];
```

3 Graphics

3.1 Gaylord's original graphics

We can create a snapshot of the entire evolution of the forest preserve using the following code:

```
Show[Graphics[RasterArray[
      Reverse[trees] /. Thread[{0, 1}->
            (Map[RGBColor, Table[Random[], {2}, {3}]]  /.
                 RGBColor[{x__}] -> RGBColor[x]) ]]], AspectRatio ->
Automatic]
```

We can create an *animation* of the evolution of the forest preserve using
CA Animation
While a static figure can be used to detect CA patterns, it is useful to watch the evolutionary process unfolding. To do this, an animation can be created.

In creating an animation using RasterArray graphics, the main thing that has to be taken into account is that the rectangles, representing site values, that re-appear in successive figures in the series that constitute the flip cards of the animation, must be the same size and color. This requires two things:

(1) each figure must have the same number of rows. This is accomplished by adding enough rows of 0's to the trees used in each figure to bring the total number of lines in the figure up to the Length of the final CA (so the entire CA computation must be completed before the graphics are created).

(2) the transformation rule for converting site values to RGBColors must be applied after all of the site values in all of the CA figures in the animation are calculated and before the graphics command is applied.

The CA animation is created using:

```
Map[Show[Graphics[RasterArray[#]],AspectRatio->Automatic]&,
     Map[Join[Table[Table[0,{n}],{Length[trees]-#}],
               Reverse[Take[trees,#]]]&,
                     Range[Length[trees]] ]/.Thread[{0, 1}->
                           (Map[RGBColor, Table[Random[],{2},{3}]]/.
                             RGBColor[{x__}]->RGBColor[x])] ];
```

(note: after creating the animation in a Mathematica notebook, double-click on the bracket enclosing all of the graphics cells (this bracket is created when Automatic Grouping has been selected in the Cell menu). This will close up the graphics cells, leaving only the first cell exposed. Double-clicking on the exposed cell starts the animation. The speed of the animation can be adjusted using the number keys (1 is the slowest speed and 9 is the fastest speed)).

3.2 Fateman's Common Windows graphics

The Lisp listing contains a program `smokey` that when executed pops up a new X-window: a rectangle titled "forest" that has a row for each generation, and a column for each tree position. The evolution of the forest sweeps up from the bottom of the window. A 500 by 500 window occupies about 1/4 of the screen of a 1000 by 1000 pixel display. As is typical for X-windows, its appearance is controlled by the defaults of the window manager, but typically this includes mechanism so the user can operate on it in various ways. For example, it can be closed to an icon, moved, or destroyed, by mouse selections on the frame (or by program control). The window is ordinarily retained indefinitely, so long as the program that originated it still exists. Several windows (all titled "forest") can be set up, enabling the comparison of windows with different parameter settings. Various "window grabber" utilities can be used to collect one or more of the windows in a file, for transmission, saving, or printing.

4 Timings

4.1 Magma times

Some timings - all on a SPARC IPC with 16 meg of memory.

```
> time a := mySmokey2( 10, 0.2, 0.4, 5 );
Time: 0.089
> time b := mySmokey2( 25, 0.2, 0.4, 50 );
Time: 2.469
> time c := mySmokey2( 100, 0.2, 0.4, 50 );
Time: 11.068
```

As a final point, this is not the kind of task Magma was designed to do. Having said that I think the code turned out quite well in the end.

(graham) Extrapolating to a 500 by 500 expanse, this would become 553 seconds. Considering the differences in speed between the HP system used for Mathematica and Lisp below, this would be perhaps 100 seconds.

4.2 SML times

```
structure Smokey0 = Smokey(val n=100 val m=10 val p=0.2 val f = 0.4);
- timing( Smokey0.mySmokey );
Non GC 0.01   GC 0.0  both 0.01 secs
val it =
  [[|0,0,1,0,1,1,1,0,0,0,1,1,...|],[|0,0,1,0,1,1,1,0,0,0,0,0,...|],
  ...
   [|0,0,0,0,0,0,0,1,1,0,0,0,...|],[|0,0,0,0,0,0,0,0,0,0,0,1,...|]]
  : int array list

structure Smokey1 = Smokey(val n=10000 val m=100 val p=0.2 val f = 0.4);
- timing( Smokey1.mySmokey );
Non GC 8.82   GC 3.62  both 12.44 secs
val it =
  [[|0,1,0,1,0,0,1,0,0,1,0,1,...|],[|0,0,0,1,0,0,1,1,0,1,0,0,...|],
   ...
   [|1,1,1,0,0,1,0,0,0,1,1,0,...|],[|0,0,0,0,0,0,0,0,0,1,1,0,...|],...]
  : int array list

structure Smokey2 = Smokey(val n=10000 val m=1000 val p=0.2 val f = 0.4);
- timing( Smokey2.mySmokey );
Non GC 101.46   GC 48.46  both 149.92 secs
val it =
  [[|1,0,0,0,1,1,1,1,0,1,1,0,...|],[|0,0,0,0,0,0,0,0,0,0,0,0,...|],
  ...
   [|1,0,1,0,1,1,1,0,0,0,0,0,...|],[|0,0,0,0,0,0,0,0,1,0,0,0,...|],...]
  : int array list
```

Based on this an interpolation to the case of 500 by 500 would be about 3.74 seconds. (No display, however)

4.3 Improved Mathematica program times

(Times on parker.eecs.berkeley.edu (HP 9000/755) using Mathematica 2.2)
Here are some typical sample times:

```
In[7]:= Timing[MySmokey[500,0.4,0.2,10];]

Out[7]= {7.78 Second, Null}

In[8]:= Timing[MySmokey[500,0.4,0.2,20];]

Out[8]= {13.84 Second, Null}
```

Presumably if we wait long enough, and if the timing remains a linear function of the number of generations. the 500 generations will appear in 346 to 389 seconds.

4.4 Lisp times

One of the advantages of using a system with a sophisticated programming and debugging environment is that one can polish code with a more intimate picture of what is going on.

In Lisp using Allegro 4.2 Common Lisp on a HP 900/735, we profiled the program invocation (quietsmokey 500 0.4 0.2 500) to find this:

```
Sample represents 2.6 seconds of processor time (out of a total of 2.6)

Times below 1.0% will be suppressed.

  %     %     self  total            self   total  Function
 Time  Cum.   secs   secs    calls ms/call ms/call   name
 51.7  51.7    1.3    1.3                          ``_qfmul''
 22.7  74.4    0.6    0.6                          TREEGROWIGNITE
 14.2  88.6    0.4    0.4                          ``_srand48''
  4.3  92.9    0.1    0.1                          ``_lrand48''
  3.3  96.2    0.1    0.1                          SPREADFIRE
```

With the profiler off, the CPU time is not 2.6 seconds, but only about 1.9 seconds. About 584 bytes of structure is used up (500 for the byte array for the forest).

Of the 5 programs mentioned by the profiler, only two are written in Lisp, and they use about 25% of the time. The rest of the time is spent in library routines concerned with the random number generation. So we can't polish our own code, we can only try to make it use the built-in subroutines less often.

To get a run even faster than that indicated above, we precomputed a large collection of random numbers and stored them in an array for re-use. We are generating, for this problem, on the order of 250,000 random numbers. If we generate say 10,003 random numbers at the time we load the system, once for all time, then the overhead does not appear in our timings, although a certain periodicity may be evident if we run the simulation for enough generations.

This replaces about 69% of the time with an initialization overhead plus a short "stepping" program, `next-lrand48` to grab the next number in an array. `(quietsmokey 500 .4 .2 500)` is about 0.55 seconds .

By changing that array to a circularly linked list, we made it even faster, at least on this system. Instead of looking at the next cell (modulo the size of the array) we followed a pointer to the next element. It gets the time down to about 0.36 seconds. Systems with other memory organizations may not experience this speedup.

As a final step, we recompiled the program in the Python research compiler developed at Carnegie-Mellon University. The CMU Common Lisp (still on the same HP computer), running the program that is re-using the random numbers in a circular list, uses only 0.18 seconds.

Dominating the time for the computation is the display time. For a 500 by 500 window it is 4.6 seconds (5.5 real time), using a remote HP server and a local X-window server.

Richard Fateman.

5 Argumentation

Once a program is written, it can be modified and re-used. A person basing a new program on one of these samples presumably would save construction time. More examples, e.g. of Conway's classic "Game of Life" would have to deal with the 2-dimensional aspects of cellular automata (note: there is a `Life.m` program in the Mathematica library.)

Given an efficient but readable module (it can be obscure, but it the only parts that need to be changed are easily changed), we have a potentially more valuable program than one that is "merely" easy to understand, but cannot be scaled up to larger problems.

Of course, some problems are not going to be scaled up, and a one-time "wasteful" expenditure of computer time is often justified if it relieves the human program of the unpleasant task of revision of a program.

In terms of this forest-fire simulation, If a run-time of 100 seconds is acceptable (without graphics!) then any of the programs given here would do. Actually, experimentation is the heart of this simulation: one run would hardly do for any but the most uninterested observer.

Since graphics is required it appears that the Lisp program among those offered here can handle large displays rapidly. Handling small arrays, and slowly, is an option with Mathematica.

6 thanks

Thanks to Richard Gaylord for suggesting this problem.

References

[1] Richard J. Gaylord and Paul R. Wellin. *Computer Simulations with Mathematica: Explorations in Complex Physical and Biological Systems.* Santa Clara, CA: Springer-Verlag TELOS, 1995. (Pages 147–149 are our example, and then the chapter goes on to deal with 2D systems.)

[2] Paczuski, Maya and Bak, Per. 1993 "Theory of the one-dimensional forest-fire model," *Phys. Rev. E (Statistical Physics, Plasmas, Fluids and Related Interdisciplinary Topics)(48)* Nov. 1993, p. R3214–R3216.

[3] Bak, Per and Paczuski, Maya. 1993 "Why Nature is Complex," *Physics World (6)* Dec. 1993,p. 39–43.

[4] Wolfram, Stephen. *Mathematica: A system for Doing Mathematics by Computer*, Addison-Wesley, 1988., 1991