Subject: Re: Can I do this without using loops?
Posted by David Ritscher on Fri, 14 Jun 1996 07:00:00 GMT
View Forum Message <> Reply to Message

--

David Ritscher
Raum 47.2.401
Zentralinstitut fuer Biomedizinische Technik
Albert-Einstein-Allee 47
Universitaet Ulm
D-89069 ULM
Germany

Tel: ++49 (731) 502 5313
Fax: ++49 (731) 502 5315
internet: david.ritscher@zibmt.uni-ulm.de

This is a comparison of different for-loop and for-loopless approaches to an indexing problem. A lot can be learned from the timing results.

Once more, regarding the original posting from S Bhattacharyya:

> Q2) I have a generic array foo=fltarr(a,b). I'd like to copy findgen(b)
> into every column. Any way of doing this without loops ?

I stand corrected - there are a couple of ways to accomplish the second of the two examples (Q2) without using for-loops. I wrote the various methods into subroutines, and testing the times for each method, using first a small size that easily fit inside memory, and then using large arrays where my machine was forced to page a lot. I ran them on an HP 715 / 64, with 96 Mbyte RAM and 320 Mbyte swapspace. I ran them both under IDL and PV-Wave, versions: IDL. Version 3.1.0 and PV-WAVE v6.01. Sometimes one was faster, sometimes the other (often by a factor of two or more!). I repeated the tests several times, and the results were quite consistent. I didn't perform any averaging, and didn't do anything to make it all particularly accurate. This should not be considered a benchmark between the two programs. My goal was just to get some feel which methods were more effective.

I compared both the problem of inserting columns (that of Q2), plus the problem of inserting rows of findgen into an array. These two problems turn out to be quite different, with different optimal solutions.

The methods compared are those proposed by:
1. Kenneth P. Bowman

2. me
3. by both David Fanning and Dan Carr (plus some recent news posters)
4. Paul C. Sorenson (paulcs@netcom.com)
5. a brute-force, double-for-loop method (to demonstrate what not to do)
6. exactly like 5 except that the long in the loop variable that is
   assigned to each element not explicitly converted to float (this
   happens explicitly).
7. as in 5., but with the loops reversed, causing thrashing
8., 9. Robert Cannon <rcc@hera.neuro.soton.ac.uk>

I wrote them each into a similar form (see the listing at the end of
this note). Here is the algorithm for each. The version written to
insert a row instead of a column is similar in form to these.
Compare, for example:
 insert rows:
  one_column = findgen(1, columns)
  for i = 0L, rows-1 do array(i, 0) = one_column
with
 insert columns:
  one_row = findgen(rows)
  for j = 0L, columns-1 do array(0, j) = one_row
All routines are listed at the end of this note.

The algorithms:
1. array = fltarr(rows, columns, /nozero)
   for j = 0L, columns-1 do array(*, j) = float(j)

2. array = fltarr(rows, columns, /nozero)
   one_row = findgen(rows)
   for j = 0L, columns-1 do array(0, j) = one_row

3. array = FINDGEN(rows) # REPLICATE(1, columns)

4. array = (findgen(1,columns))(bytarr(rows),*)

5. array = fltarr(rows, columns, /nozero)
   for j = 0L, columns-1 do $
      for i = 0L, rows-1 do $
  array(i, j) = float(i)
6. as in 5., but '= i' instead of '= float(i)'
7. as in 5., but reversed order of the two for statements

8. array = rebin(findgen(1, columns), rows, columns)

9. array = transpose(findgen(columns, rows) mod columns)

Here are the execution times, in seconds, under IDL and PV-Wave, for the specified array sizes, for each of the algorithms above. The left two columns show the times for inserting columns, the second two columns involve inserting rows.

Tested using the following number of rows and columns: 3000, 3000

	Insert columns		Insert rows	
	IDL	PV-Wave	IDL	PV-Wave
1.	22.84	17.58	30.48	21.31
2.	22.77	30.81	4.45	1.48
3.	3.88	9.73	3.87	9.70
4.	12.05	11.93	5.67	7.17
5.	89.24	164.56	87.51	163.15
6.	141.93	385.72	140.45	389.53
7.	92.75	166.44	92.82	169.71
8.	1.30	4.96	9.78	10.29
9.	30.75	28.71	18.40	21.60

Tested using the following number of rows and columns: 1000, 30000

	Insert columns		Insert rows	
	IDL	PV-Wave	IDL	PV-Wave
1.	84.95	71.09	Inf	Inf
2.	Inf	Inf	28.52	22.69
3.	27.65	42.95	30.88	50.10
4.	129.53	138.84	124.39	126.26
5.	300.62	547.51	296.74	556.55
6.	-	1284.83	-	1300.17
7.	Inf	Inf	Inf	Inf
8.	20.07	28.76	Inf	Inf
9.	-	3100.02	158.07	161.23

Where here I poetically define "Inf" to be anything that takes longer than a day (I suspect these would all end up taking about 10^4 seconds).

A few conclusions can be drawn from the timings:

The most important thing is to differentiate between problems that fit fully within physical memory (RAM) and those that require the use of virtual memory. For example, on my 96 Mbyte machine the first example of using arrays of 3000x3000 floats (= ~36 Mbytes) fit within my machine. However, increasing the problem to 1000x30000 (= ~120 Mbytes) makes it exceed my machines RAM, and it becomes necessary for my machine to swap the array in and out of memory as it works through the elements. This is a very slow operation, that will almost always be the dominant

factor in determining how long the routine takes. Although this problem is only about three times larger than the original, the fastest solutions were 10 times longer. Some solutions, that had taken 20 seconds with the first problem, were still not finished after the machine had run for a full day.

For problems that do not fit into physical memory it is crucial that all operations scan over the leftmost dimension first (i.e. rows) before later dimensions (columns, etc.). The statement array(i, *) = i inside a for-loop causes the machine to scan through the second dimension before the first dimension. That turned this seemingly simple statement into a problem that was not finished a day later. The slow approach of using double for-loops was many times faster than this more elegant approach when the for-loops were done in the proper order:

```
    for j=0L, rows-1 do for i=0L, columns-1 do array(i, j) = ...
```

but reversing the order of the for-loops turned it into another problem that was not finished a day later. The best solution of problems that are inherently row-oriented (such as inserting something into each row of a matrix) are different from the best solution of column-oriented problems. These optimal solutions ended up being the same solutions that were optimal for the problem that fit within physical memory.

The only solution that performed pretty well for both row-oriented and column-oriented problems was the elegant cross-product approach (approach 3.), despite the fact that it requires an extra multiply for each array element: FINDGEN(rows) # REPLICATE(1, columns)

Comparing the times of solutions 5. and 7., one notes that placing the for-loops in the wrong order has very little effect on execution time as long as the problem fits within physical memory, but has a drastic effect when that is not the case.

The optimal solution to the problem of inserting findgen into rows of the array ended up being the for-loop looping over the columns, solution 2.:

```
  one_row = findgen(rows)
  for j = 0L, columns-1 do array(0, j) = one_row
```

The optimal solution for the problem of inserting findgen into the columns was accomplished with rebin, solution 8.:

```
  array = rebin(findgen(1, columns), rows, columns)
```

The clever vector indexing approach, solution 4.:

```
  (findgen(1,columns))(bytarr(rows),*)
```

performed surprisingly poorly. I can't explain why; it's simply a function of how the interpreter functions with such an indexing scheme.

It was interesting to note the burden that a function call creates when it is inside an inner for-loop. In solution 5. an implicit type

conversion is made from long integer to float: array(i, j) = i
when this is changed to array(i, j) = float(i) (solution 6.), the
execution times are much slower, sometimes more than doubly slow.

One can see from the execution times that at least part of the code for
IDL and PVWave are diverging (presumably the memory administration and
indexing systems, and perhaps the interpreter); often one language is
twice as fast as the other in a given solution, but then the other
language is twice as fast on another problem. Often the times are
essentially identical. It's clear that neither company has set a goal
of perfecting execution speed under all conditions.

As I mentioned, when one does operations along a higher dimension
first for a problem that doesn't fit into memory, the problem becomes
very slow. I was curious what was happening, because after two days I
still hadn't gotten the result from one of these. So I added an extra
print statement to my test routine test1_rows.pro to look at the
timing as it inserts each row. The best time for this example was
about 20 seconds, but each of the 1000 rows took longer than this, due
to the thrashing to swap memory in and out. Here are the timings, in
seconds for the first few row computations. The time per row keeps
growing, but always slower:

test1_rows, 1000, 30000

row	elapsed time	average time/row
1	13.296000	13.327000
2	81.090000	40.560500
3	150.33500	50.121667
4	218.95900	54.750000
5	294.14200	58.832600
6	365.92100	60.992000
7	435.88800	62.274143
8	505.99300	63.253000
9	579.73700	64.418556
10	649.93200	64.997300
11	721.52000	65.595636
12	790.99900	65.919167
13	863.28500	66.408923
14	936.90500	66.923929
15	1008.2590	67.219333
16	1080.0630	67.505813
17	1154.3780	67.906412
18	1227.2110	68.180111
19	1297.6890	68.301053
20	1369.7120	68.487150
21	1444.2740	68.776429
22	1517.4780	68.977636

23 1589.7140 69.119348

I'm guessing that completion will take something like 100,000 seconds, or about 5,000 times longer than the best time for this solution.

Morals of the story:
A good optimized C compiler worries about a lot of the details of making these things efficient. IDL / PVWave are not designed to optimize the problem for you, so in the case where time becomes crucial, you have to do the optimizing:

1. Avoiding accessing in order of higher dimension first (i.e., columns before rows).
2. Try to avoid an inner for-loop - vectorize the fastest-changing index when possible.
3. Avoid function calls within an inner for-loop. (but this falls under the last, avoid the inner for-loop in general).
4. Careful choice of approach can then optimize a problem between different fairly effective approaches. Interestingly, these solutions can be different between IDL and PV-Wave. One just has to experiment.

****

The same method used in (4) can be applied to the first question:

> Q1) I have a generic array foo(x,y). I'd like to divide each column
> by its max. Can this be done without looping ?
>

And provides a solution where the main looping can be done without a for-loop. I still see no way to avoid using a for-loop to get the maximums for each column.

```
; make an example array:
rows = 3
columns = 5
foo = findgen(rows, columns)
; create a vector of 1 / maxes for each column:
inv maxes = fltarr(rows)
for i = 0L, rows-1 do inv_maxes(i) = 1. / max(foo(i, *))
;
; Now perform the scaling, scanning through the elements of 'foo' and
; repeatedly through the inv_maxes vector:
foo = foo * inv_maxes(*, bytarr(columns))
```

I didn't check the times on this, but assume it will unfortunately also, like solution 4., perform slowly.

; Here are the actual routines I used for the time testing:

```
PRO TEST1, rows, columns
strt = systime(1)
;
array = fltarr(rows, columns, /nozero)
for j = 0L, columns-1 do array(*, j) = j
;
print, 'elapsed time: ',systime(1) - strt
return
END


PRO TEST2, rows, columns
strt = systime(1)
;
array = fltarr(rows, columns, /nozero)
one_column = findgen(1, columns)
for i = 0L, rows-1 do array(i, 0) = one_column
;
print, 'elapsed time: ',systime(1) - strt
return
END


PRO TEST3, rows, columns
strt = systime(1)
;
array =  replicate(1.0, rows) # findgen(1, columns)
;
print, 'elapsed time: ',systime(1) - strt
return
END


PRO TEST4, rows, columns
strt = systime(1)
;
array = (findgen(1,columns))(bytarr(rows),*)
;
print, 'elapsed time: ',systime(1) - strt
return
END
```

```
PRO TEST5, rows, columns
strt = systime(1)
;
array = fltarr(rows, columns, /nozero)
for j = 0L, columns-1 do $
   for i = 0L, rows-1 do $
      array(i, j) = j
;
print, 'elapsed time: ',systime(1) - strt
return
;
END


PRO TEST6, rows, columns
strt = systime(1)
;
array = fltarr(rows, columns, /nozero)
for j = 0L, columns-1 do $
   for i = 0L, rows-1 do $
      array(i, j) = float(j)
;
print, 'elapsed time: ',systime(1) - strt
return
;
END


PRO TEST7, rows, columns
strt = systime(1)
;
array = fltarr(rows, columns, /nozero)
   for i = 0L, rows-1 do $
for j = 0L, columns-1 do $
      array(i, j) = j
;
print, 'elapsed time: ',systime(1) - strt
return
;
END


PRO TEST8, rows, columns
strt = systime(1)
;
array = rebin(findgen(1, columns), rows, columns)
;
print, 'elapsed time: ',systime(1) - strt
```

```
return
;
END


PRO TEST9, rows, columns
strt = systime(1)
;
array = transpose(findgen(columns, rows) mod columns )
;
print, 'elapsed time: ',systime(1) - strt
return
;
END



; the following are similar to the above, but insert rows intead of
; columns of 'findgen's:

PRO TEST1_rows, rows, columns
strt = systime(1)
;
array = fltarr(rows, columns, /nozero)
for i = 0L, rows-1 do array(i, *) = i
;
print, 'elapsed time: ',systime(1) - strt
return
END


PRO TEST2_rows, rows, columns
strt = systime(1)
;
array = fltarr(rows, columns, /nozero)
one_row = findgen(rows)
for j = 0L, columns-1 do array(0, j) = one_row
;
print, 'elapsed time: ',systime(1) - strt
return
END


PRO TEST3_rows, rows, columns
strt = systime(1)
;
array =  findgen(rows) # replicate(1.0, 1, columns)
;
```

```
print, 'elapsed time: ',systime(1) - strt
return
END


PRO TEST4_rows, rows, columns
strt = systime(1)
;
array = (findgen(rows))(*, bytarr(columns))
;
print, 'elapsed time: ',systime(1) - strt
return
END


PRO TEST5_rows, rows, columns
strt = systime(1)
;
array = fltarr(rows, columns, /nozero)
for j = 0L, columns-1 do $
  for i = 0L, rows-1 do $
    array(i, j) = i
;
print, 'elapsed time: ',systime(1) - strt
return
;
END


PRO TEST6_rows, rows, columns
strt = systime(1)
;
array = fltarr(rows, columns, /nozero)
for j = 0L, columns-1 do $
  for i = 0L, rows-1 do $
    array(i, j) = float(i)
;
print, 'elapsed time: ',systime(1) - strt
return
;
END


PRO TEST7_rows, rows, columns
strt = systime(1)
;
array = fltarr(rows, columns, /nozero)
  for i = 0L, rows-1 do $
```

```
for j = 0L, columns-1 do $
      array(i, j) = i
;
print, 'elapsed time: ',systime(1) - strt
return
;
END


PRO TEST8_rows, rows, columns
strt = systime(1)
;
array = rebin(findgen(rows), rows, columns)
;
print, 'elapsed time: ',systime(1) - strt
return
;
END


PRO TEST9_rows, rows, columns
strt = systime(1)
;
array = findgen(rows, columns) mod rows
;
print, 'elapsed time: ',systime(1) - strt
return
;
END
```

File Attachments

1) idl2.txt, downloaded 126 times