Subject: LIST performance
Posted by JDS on Sat, 06 Nov 2010 21:07:02 GMT
View Forum Message <> Reply to Message

One of the performance bottlenecks IDL users first run into is the deficiencies of simple-minded accumulation. That is, if you will be accumulating some unknown number of elements into an array throughout some continued operation, simple methods like:

```
foreach thing,bucket_o_things,i do begin
   stuff=something_which_produces_an_unknown_number_of_element( thing)
  if n_elements(array) eq 0 then array=stuff else array=[array,stuff]
endforeach
```

fail horribly. The problem here is the seemingly innocuous call "array=[array,stuff]," which 1) makes a new list which can fit both pieces, and 2) copies both pieces in. This results in a *huge* amount of wasted copying. To overcome this, a typical approach is to preallocate an array of some size, filling it until you run out room, at which point you extend it by some pre-specified block size. It's also typical to double this block size each time you make such an extension. This drastically reduces the number of concatenations, at the cost of some bookkeeping and "wasted" memory allocation for the unused elements which must be trimmed off the end.

At first glance, it would seem the LIST() object could save you all this trouble: just a make a list, and "add" 'stuff' to it as needed, no copying required. Unfortunately, the performance of LISTs for accumulation, while much better than simple-minded accumulation as above, really can't compete with even simple array-expansion methods. See below for a test of this.

Part of the problem is that the toArray method cannot operate on list elements which are arrays. Even without this, however, LIST's performance simply can't match a simple-minded "expand-and-concatenate" accumulation method. In fact, even a pointer array significantly outperforms LIST (though it's really only an option when you know in advance how many accumulation iterations will occur... not always possible). Example output:

```
EXPAND-CONCATENATE accumulate:      0.19039917
PTR accummulate:                  0.40397215
LIST accummulate:                  1.5151551
```

I'm not sure why this is. In principle, a lightweight, (C) pointer-based linked list should have very good performance internally. So, while very useful for aggregating and keeping track of disparate data types, LIST's are less helpful for working with large data sets.

JD

```
++++++++++++++
n=100000L

;; First method: Expand array in chunks, doubling each time.

t=systime(1)
bs=25L
off=0
array=lonarr(bs,/NOZERO)
sarr=bs
for i=0L,n-1 do begin
  len=1+(i mod 100)
  if (off+len) ge sarr then begin
    bs*=2
    array=[array,lonarr(bs,/NOZERO)]
    sarr+=bs
  endif
  array[off]=indgen(len)
  off+=len
endfor
array=array[0:off-1]
print,'EXPAND-CONCATENATE accummulate: ',systime(t)-t

;; Second method: Use pointers
parr=ptrarr(n)
c=0
for i=0L,n-1 do begin
  len=1+(i mod 100)
  parr[i]=ptr_new(indgen(len))
  c+=len
endfor

new=intarr(c,/NOZERO) ;; exactly the correct size
off=0L
foreach elem,parr do begin
  new[off]=*elem
  off+=n_elements(*elem)
endforeach
print,'PTR accumulate:            ',systime(1)-t

;; Third method: Use LIST
t=systime(1)
list=list()
c=0
```

```
for i=0L,n-1 do begin
   len=1+(i mod 100)
   list.add,indgen(len)
   c+=len
endfor

;; List::ToArray should do this for you internally!!!
new2=intarr(c,/NOZERO) ;; exactly the correct size
off=0L
foreach elem,list do begin
   new2[off]=elem
   off+=n_elements(elem)
endforeach
print,'LIST accummulate:               ',systime(1)-t

END
```