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
```

---

Subject: Re: LIST performance
Posted by chris_torrence@NOSPAM on Fri, 12 Nov 2010 17:46:01 GMT
View Forum Message <> Reply to Message

Hi JD,

Just to put in my 2 cents, LIST (like much of IDL) is a general purpose set of functionality. It was never designed to replace the use of arrays, where the data is laid out in memory in the most efficient manner. The IDL LIST is implemented as a doubly-linked list, which is very efficient for adding & removing elements from arbitrary positions, especially elements of different or complicated types. If you're only using integers or floats, you probably don't want to use a list.

Here's an example for adding & removing random elements from an array of structures:

```
print,'          N         Array(s)' + $
  '       List(s)        Ratio'
for nn=2,4 do begin
  n = 10L^nn
  iter = 10L^(6 - nn)
  a = REPLICATE(!map, n)
  t = systime(1)
  for i=0,iter-1 do begin
    index = RANDOMU(seed,1)*(n-2) + 1
    a = [a[0:index-1], !map, a[index:*]]
    a = [a[0:index-1], a[index+1:*]]
  endfor
```

```
  timearray = systime(1)-t

  a = LIST(REPLICATE(!map, n), /EXTRACT)
  t = systime(1)
  for i=0,iter-1 do begin
    index = RANDOMU(seed,1)*(n-2) + 1
    a.Add, !map, index
    a.Remove, index
  endfor
  timelist = systime(1)-t

  print, n, timearray, timelist, timearray/timelist
endfor
end
```

When I run this on my Win32 XP laptop, I get the following results:

| N | Array(s) | List(s) | Ratio |
|---|---|---|---|
| 100 | 3.3750000 | 0.38999987 | 8.6538491 |
| 1000 | 5.5160000 | 0.10899997 | 50.605520 |
| 10000 | 7.2500000 | 0.078000069 | 92.948636 |

Part of the reason the LIST is slower in your example is the overhead with both error checking and the operator overloading. The List::Add and List::overloadForeach are both method calls, so there is some additional overhead for making a call instead of just doing it in-place like in your pointer example.

Now, all that being said, we'll continue to make performance improvements in future versions. For example, I just rewrote the List::ToArray to be about 10 times faster. It's still "brutally slow" for your particular example, but for other scenarios it's much faster.

Finally, I like your idea about making the ::ToArray method work properly for list elements that are arrays. I'll see what I can do.

Keep the feedback coming.
Cheers,

Chris
ITTVIS

---

Subject: Re: LIST performance
Posted by JDS on Thu, 16 Dec 2010 23:32:37 GMT
View Forum Message <> Reply to Message

On Nov 12, 12:46 pm, Chris Torrence <gorth...@gmail.com> wrote:

> Hi JD,
>
> Just to put in my 2 cents, LIST (like much of IDL) is a general
> purpose set of functionality. It was never designed to replace the use
> of arrays, where the data is laid out in memory in the most efficient
> manner. The IDL LIST is implemented as a doubly-linked list, which is
> very efficient for adding & removing elements from arbitrary
> positions, especially elements of different or complicated types. If
> you're only using integers or floats, you probably don't want to use a
> list.

Thanks, Chris. I suppose having learned to program in C, "linked-list"
implies some raw, close-to-the-hardware speed, i.e. more akin to normal
IDL array operations than to object-wrapped IDL pointers referencing
small data sets ;). But I agree that LIST has a lot of unnecessary
overhead if all you want is an expandable array of a given data type.
The problem is... IDL has no such thing as flexible array expansion, so
we use various tricks for this commonly needed storage pattern.

In any case, I did expand my analysis of array accumulation, varying the
chunk size and the total number of accumulations over wide ranges. I
tested four algorithms: "expanding concatenation", "pre-allocated
pointer array", "LIST", and a hybrid approach, adding pointers to each
chunk to a list. Here is the result:

http://idlwave.org/idl/accumulate.png

and code:

http://idlwave.org/idl/test_list_accum.pro

For small chunk sizes (1 integer added per accumulation step, dotted
lines), LIST is very inefficient, the POINTER method is several times
slower than a "doubling concatenation". As you increase the total
amount accumulated per step however, the story changes. For large
chunks, on average 5000/2=2500 items per accumulation step, the POINTER
method is the clear favorite, and LIST beats out my hacked doubling
concatenation, likely because it is less efficient with memory. Still,
you might hope that LIST and an array of POINTERS would offer (more)
similar performance.

> Here's an example for adding & removing random elements from an array

> of structures:
>
<snip>
> a = [a[0:index-1], !map, a[index:*]]
> a = [a[0:index-1], a[index+1:*]]
<snip>

Yes, random insertion is clearly where a linked list
would excel vs. rote recopying of a large array on every insertion. So
would it have had you written your own linked list using IDL PTRs (as
I
believe a few people have). It's for this reason that I feel that
random *accumulation* is a better test of LIST's overall speed, since
LIST is a native, internal IDL type and can potentially do more than
we
could do.

> Part of the reason the LIST is slower in your example is the overhead
> with both error checking and the operator overloading. The List::Add
> and List::overloadForeach are both method calls, so there is some
> additional overhead for making a call instead of just doing it in-
> place like in your pointer example.

Right. That's clearly evident in the comparison between LIST and
Pointer (Blue/Green) in my results. I'm a bit surprised that the
overhead would still be significant when adding thousands of elements
at
a time.

> Now, all that being said, we'll continue to make performance
> improvements in future versions. For example, I just rewrote the
> List::ToArray to be about 10 times faster. It's still "brutally slow"
> for your particular example, but for other scenarios it's much faster.
>
> Finally, I like your idea about making the ::ToArray method work
> properly for list elements that are arrays. I'll see what I can do.

Thanks. I had presumed there would be plenty of optimization overhead
left for LIST. Good luck squeezing all of it out!

JD