Subject: Re: LIST performance
Posted by penteado on Sat, 06 Nov 2010 21:55:46 GMT
View Forum Message <> Reply to Message

On Nov 6, 7:07 pm, JD Smith <jdtsmith.nos...@yahoo.com> wrote:
> 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.

Do you have the results as a function of number of elements? The curves will have different shapes, and the expected behavior might occur only on some ranges of values. For one thing, expand-concantenate is not a smooth function, and it also depends on another parameter (the size of the buffer).

There is one plot of that kind in Michael Galloy's post:

http://michaelgalloy.com/2010/07/22/idl-8-0-lists-and-hashes .html

Subject: Re: LIST performance
Posted by JDS on Sun, 07 Nov 2010 00:03:54 GMT
View Forum Message <> Reply to Message

On Nov 6, 3:55 pm, Paulo Penteado <pp.pente...@gmail.com> wrote:
> On Nov 6, 7:07 pm, JD Smith <jdtsmith.nos...@yahoo.com> wrote:
>
>> 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.
>
> Do you have the results as a function of number of elements? The
> curves will have different shapes, and the expected behavior might
> occur only on some ranges of values. For one thing, expand-
> concantenate is not a smooth function, and it also depends on another
> parameter (the size of the buffer).

Good point; I ran for different number of accumulation steps

(resulting in increasingly larger final arrays):

100	7.8164026
1000	6.8048671
10000	6.2041477
25000	6.2937977
50000	6.3610957
100000	6.6396416
500000	5.7915670

The right column is the time for LIST relative to expand/concatenate accumulation, adding ~50 numbers at a time. It does not seem to depend on the number of accumulation steps.

The relative performance does, however, depend on how much you accumulate per step. If I keep the final accumulated array fitting in memory to avoid dependence on swapping, I find that accumulating large chunks at a time favors the PTR method and the LIST method over the concatenation approach, such that for very large chunks (few thousand added per iteration, or more), both are faster than the "expand" style. Still, never is LIST faster than PTR accumulation; I wouldn't even be surprised if LISTs used IDL PTRs internally.

> There is one plot of that kind in Michael Galloy's post:
>
> http://michaelgalloy.com/2010/07/22/idl-8-0-lists-and-hashes .html

Thanks, I hadn't seen this. It did turn me on to the EXTRACT keyword, which in principle, coupled with toArray(), would make this a trivial operation. Sadly, the latter is freakishly slow compared to the simple "count, offset, and insert" method I had used to pull the data out of the LIST (also for the PTRARR). My guess is it's doing something akin to the first example in my former post: recopying everything on each add (though the main penalty comes during toArray(), so perhaps each element on the LIST is somewhat heavyweight).

So, distilling it down:

For accumulating hundreds or fewer elements any number of times, an expand/concatenate approach is favored. Otherwise, use pointers. LIST is slightly more convenient, but in its most convenient form (using /EXTRACT and toArray()), it is brutally slow. Pointers work best when you know how many slots to preallocate (i.e. how many steps in your accumulation); if you don't, LIST may work for you (or, for the best of all worlds, just use an expand/concatenate approach with a PTRARR).

Perhaps Michael can include an "expanding concatenation" curve to his graph. For his case of adding one element at a time (not so commonplace?), I predict it will blow the others out of the water (as would a simple PTR-based accumulate).

I'm still trying to decide how useful LIST and HASH are.

JD

Subject: Re: LIST performance
Posted by penteado on Sun, 07 Nov 2010 01:31:34 GMT
View Forum Message <> Reply to Message

On Nov 6, 10:03 pm, JD Smith <jdtsmith.nos...@yahoo.com> wrote:
> I'm still trying to decide how useful LIST and HASH are.

I am surprised they are that slow. But that was not an issue in most (maybe all) of the many uses I had of them in the past few months. I had been wanting lists and hashes for a long time, and was starting to move to Python for their lack in IDL.

It is too much work, or too awkward, to store and retrieve variable or heterogeneous things using only arrays, pointers and structures. Most of the time they are not so large that performance is an issue, much more important is that the code is short and clean. Which is why lists and hashes are present in every(?) decent modern language.

One simple example is how much nicer histogram's reverse indices become with lists (and !null):

http://www.ppenteado.net/idl/histogram_pp.html

A more complicated example is the use of lists and hashes to set and retrieve multiple properties in the class I made to provide a plot grid (like multiplot does for DG):

http://www.ppenteado.net/idl/pp_multiplot__define.html

Properties can be retrieved or set as lists, with one element for each column/line/plot. That way, for instance, xtickvalues can be a list with arrays of different lengths for the columns. And hashes can store properties by their names. That is just for the API. Internally, the code would be a horrible mess without lists and hashes to store things that are heterogeneous and variable (in number of elements and shape/ type of the elements).

The current lists and hashes can (and should) mature in future

versions, becoming more efficient, perhaps providing some specialized subclasses (like homogeneous lists). This can happen by reimplementation or inheritance and overloading, in either IDL code or DLMs.

For instance, a derived list could be made to implement the expand-concatenate algorithm, providing the same interface, in an easy way to implement, as only a few methods would have to be written. All the classes I had made for this kind of functionality before IDL 8 were incomplete, as it was too much work to implement every relevant method, with all the needed error-checking, documentation and testing. Now, inherited classes can be made, a lot less work.

Other algorithms can even make use of the built-in lists: when the buffer gets full, instead of concatenating arrays, a new buffer array is made, and added to a list where each element is a buffer array. At the end, these arrays get put together into a flat array.

Java is an example where a bunch of different implementations are provided in the standard library, which through inheritance keep the same API for each kind of collection. For instance, it has 5 types of lists, and 9 types of maps (hashes, in IDL's nomenclature). Given that IDL's lists and hashes are regular classes, inheritance and overloading allow to easily make a similar variety of containers.

Subject: Re: LIST performance
Posted by Mark[1] on Sun, 07 Nov 2010 20:14:16 GMT
View Forum Message <> Reply to Message

A while back I wrote an MGH_Vector class that performs much the same function as IDL 8's LIST (random-access container for heterogeneous data). For the simple problem of accumulating a bunch of n floats then saving the results to an array, the MGH_Vector is slower than simple array concatenation for small n and faster for large n, as you'd expect, the crossover occurring at around n = 25,000. In some quick and dirty tests I just ran, MGH_Vector outperformed LIST by a factor of 2.4 from n = 10^4 to 10^6, then at n = 10^7 MGH_Vector stayed linear, but LIST started working the paging file and then ran out of memory. (To be fair to the LIST class, the amount of memory that IDL 8 has available in GUI mode on win32 is pitifully small at 512 MiB. I could switch to the console mode, but I'm afraid I can't be bothered.)

You'd think the ITT programmers could do a better job of programming an extensible list class than little old me.

Subject: Re: LIST performance
Posted by Paul Van Delst[1] on Mon, 08 Nov 2010 15:33:22 GMT
View Forum Message <> Reply to Message

Hello,

Only indirectly related to your post re: list() performance....

FWIW, I altered some of my code recently from using structures containing PTRARRs to accumulate arrays of disparate
things to using LISTs. The latter code with the LISTs was *much* easier to understand (and I mean a *lot* easier), but
was noticeably slower than the code with my structure/PTRARR data object abomination.

IMPORTANT NOTE: To be fair my timing results are probably not worth the electrons used to display them on my screen but
they were reliably fractions of a second (at most 0.01-0.1s) with the structure/PTRARR setup, as opposed to several
seconds (5-6s) using the LISTs. Multiply that by several datasets, as well as multiple runs for unit tests, and the
difference borders on tiresome but leaning strongly towards annoying. :o)

I stuck with the slower LISTs because a) of easier code maintenance and b) I am assuming the list performance will be
improved in future versions of IDL -- my conversion was done and tested with 8.0... I haven't tested with 8.0.1. [*]

I'll need to do a bit of digging in our repository to pull out the old code and document the comparison so as make this
post more fact than hearsay.

cheers,

paulv

[*] Some earlier implementations of Fortran90 compilers had similar issues with array syntax over DO loops. That is,
given array variables like
```
     REAL, DIMENSION(100) :: a, b, c
```
operations using array syntax, like
```
  a = b + c
```
were much slower than the usual do loop:
```
  DO i = 1, 100
    a(i) = b(i) + c(i)
  END DO
```
The compilers eventually caught up performance-wise, but it took several years for the "Fortran90 is waaaay slower than
FORTRAN77" perception to dissipate.

JD Smith wrote:
> 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 Mark Piper on Tue, 09 Nov 2010 02:18:54 GMT
View Forum Message <> Reply to Message

On Nov 6, 2:07 pm, JD Smith <jdtsmith.nos...@yahoo.com> wrote:
> 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
```

This is good timing! On Wednesday, I'm giving a web seminar on using arrays, structures, lists & hashes in IDL. My webinar is pitched at an introductory level, but I do plan to show some simple performance results. I haven't put in the amount of research that JD, Paulo, Mark and Paul have shown in this thread, but I'll refer to the discussion in this thread in the webinar.

I'm doing the webinar live three times on Wednesday, November 10. The times (all local) are: 11 am Singapore, 2 pm London and 2 pm New York.

Please check the VIS website for signup information:

http://www.ittvis.com/EventsTraining/LiveWebSeminars.aspx

The webinars are recorded, so even if you can't attend a live session, please sign up and you'll receive a message when the recording is posted to our website. I also have examples that I'll use in the webinar; these can be downloaded from:

http://bit.ly/IDL-webinar-files

They'll be ready a few hours before the first webinar.

mp

Subject: Re: LIST performance
Posted by penteado on Tue, 09 Nov 2010 15:10:22 GMT
View Forum Message <> Reply to Message

Was an announcement sent for that? I just checked that the last I received was on Oct/20, for an ENVI SARscape webinar.

The timing is handy for me as well, as I was just preparing a lecture on the subject.

On Nov 9, 12:18 am, Mark Piper <mpi...@ittvis.com> wrote:
> This is good timing! On Wednesday, I'm giving a web seminar on using
> arrays, structures, lists & hashes in IDL. My webinar is pitched at an
> introductory level, but I do plan to show some simple performance
> results. I haven't put in the amount of research that JD, Paulo, Mark
> and Paul have shown in this thread, but I'll refer to the discussion
> in this thread in the webinar.
>
> I'm doing the webinar live three times on Wednesday, November 10. The
> times (all local) are: 11 am Singapore, 2 pm London and 2 pm New York.
> Please check the VIS website for signup information:
>

> http://www.ittvis.com/EventsTraining/LiveWebSeminars.aspx
>
> The webinars are recorded, so even if you can't attend a live session,
> please sign up and you'll receive a message when the recording is
> posted to our website. I also have examples that I'll use in the
> webinar; these can be downloaded from:
>
> http://bit.ly/IDL-webinar-files
>
> They'll be ready a few hours before the first webinar.
>
> mp

Subject: Re: LIST performance
Posted by Jeremy Bailin on Sat, 13 Nov 2010 22:21:23 GMT
View Forum Message <> Reply to Message

On Nov 8, 9:18 pm, Mark Piper <mpi...@ittvis.com> wrote:
> On Nov 6, 2:07 pm, JD Smith <jdtsmith.nos...@yahoo.com> wrote:
>
>
>
>
>
>
>
>
>
>
>> 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
>
>  This is good timing! On Wednesday, I'm giving a web seminar on using
>  arrays, structures, lists & hashes in IDL. My webinar is pitched at an
>  introductory level, but I do plan to show some simple performance
>  results. I haven't put in the amount of research that JD, Paulo, Mark
>  and Paul have shown in this thread, but I'll refer to the discussion
>  in this thread in the webinar.
>
```

> I'm doing the webinar live three times on Wednesday, November 10. The
> times (all local) are: 11 am Singapore, 2 pm London and 2 pm New York.
> Please check the VIS website for signup information:
>
> http://www.ittvis.com/EventsTraining/LiveWebSeminars.aspx
>
> The webinars are recorded, so even if you can't attend a live session,
> please sign up and you'll receive a message when the recording is
> posted to our website. I also have examples that I'll use in the
> webinar; these can be downloaded from:
>
> http://bit.ly/IDL-webinar-files
>
> They'll be ready a few hours before the first webinar.
>
> mp

Any idea when the archived version will be up? I couldn't make it.

-Jeremy.

Subject: Re: LIST performance
Posted by Mark Piper on Tue, 16 Nov 2010 00:29:56 GMT
View Forum Message <> Reply to Message

On Nov 13, 3:21 pm, Jeremy Bailin <astroco...@gmail.com> wrote:
> Any idea when the archived version will be up? I couldn't make it.
>
> -Jeremy.

Hi Jeremy,

The recording is up at

  http://www.ittvis.com/EventsTraining/RecordedWebinars.aspx

mp
