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