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