Subject: Dynamically resizing arrays
Posted by Jonathan Greenberg on Sat, 05 Feb 2005 00:24:00 GMT
View Forum Message <> Reply to Message

I was hoping to get some feedback on the best way of creating a "database" -- an array of fixed columns but unknown number of rows which will be appended to within some sort of loop. What is the best way of doing this?

--j

Subject: Re: Dynamically resizing arrays
Posted by Michael Wallace on Wed, 09 Feb 2005 20:02:29 GMT
View Forum Message <> Reply to Message

> Adding a row at a time requires time and memory to grow as the square
> of the size.
> Memory grows because the new chunk cannot reuse the old one, it is too
> big.
> Adding a chunk at a time reduces the coefficient by 100 or whatever.
> Doubling increases time logarithmically.
> This square growth can noticibly slow a program that does a lot of it.

Doubling reduces overall time at the expense of memory. The chunk approach tracks memory much closer at the expense of time. While a lot of square growth can noticeably slow a program, it will still be much faster than the chunk approach.

-Mike

Subject: Re: Dynamically resizing arrays
Posted by JD Smith on Mon, 21 Feb 2005 17:53:33 GMT
View Forum Message <> Reply to Message

On Wed, 09 Feb 2005 14:02:29 -0600, Michael Wallace wrote:

>> Adding a row at a time requires time and memory to grow as the square
>> of the size.
>> Memory grows because the new chunk cannot reuse the old one, it is too
>> big.
>> Adding a chunk at a time reduces the coefficient by 100 or whatever.
>> Doubling increases time logarithmically.
>> This square growth can noticibly slow a program that does a lot of it.
>
> Doubling reduces overall time at the expense of memory. The chunk
> approach tracks memory much closer at the expense of time. While a lot

> of square growth can noticeably slow a program, it will still be much
> faster than the chunk approach.

Almost any allocation algorithm is better than row-at-a-time, in terms of both time and total memory allocated. The killer here is really the number of times you allocate an N+delta_N chunk of memory, and copy the original array of size N into it. The fewer the better. In case you are wondering, this is exactly how IDL's native array concatenation works. If you have a huge array of around your total physical memory size, (1GB for me), like so:

IDL> a=make_array(1000L*1000L*1000L,VALUE=0b)

and try to add just one measly byte to it:

IDL> a=[temporary(a),1b]

your machine will at best grind to a halt as data is paged to disk, or more likely you'll get an out of memory condition, just for that one byte! Why? IDL is forced to allocate 1 billion and 1 bytes of new memory, copy those billion zeroes over, and then add the 1b to the end.

Doubling the added chunk in each round reduces execution time at the expense of *maximum* memory allocated at any given time, not total memory allocated over the full run of the program (the latter being essentially equivalent to the run time).

We did discuss this a bit a while back. See:

http://groups-beta.google.com/group/comp.lang.idl-pvwave/msg /b0155a19469fd00f

In one example involving an (a priori unknown) final array size of 100 units, row-at-a-time allocated 5050 units of memory, whereas the doubling algorithm allocated only 220. Examples like this are common place in my usage of IDL.

One good option if you worry about running into a memory ceiling is to use a doubling (or any geometric factor, e.g. 1.5) up to some maximum memory size, and then scale back to a linear, constant chunk size increment (perhaps the next to last chunk size). This gives you speed when you have the memory to spare, and preserves memory when you're running out.

JD