Subject: Chunk Array Decimation
Posted by JD Smith on Tue, 01 Oct 2002 19:32:54 GMT
View Forum Message <> Reply to Message

Histogram lovers (and haters alike):

Carsten put this question to me, and I've wasted too much time on it not to report the results. It's a deceptively simple-seeming problem.

You have a vector of indices, `inds', and a data vector `data' of the same length. The `inds' vector contains a list of many repeated indices, e.g.:

```
[4,2,6,4,1,5,6,6,3,1,7,2]
```

The job is to generate a result vector `vec' of length max(inds)+1, such that for each index i:

```
vec[i]=total(data[where(inds eq i)])
```

or zero otherwise. That is, gather all data with a given corresponding index, and total them together into the result vector at that index. If no indices are repeated, you can of course just use simple assignment, but that's a much simpler problem. Here's a very straightforward implementation based on this idea:

```
mx=max(inds)
vec1=fltarr(mx+1)
for j=0L,mx do begin
  wh=where(inds eq j,cnt)
  if cnt eq 0 then continue
  vec1[j]=total(data[wh])
endfor
```

This is slow. Very slow. Verrrrrrrrrry slow. And wasteful. You search through the index vector using where() many, many times. Beware of constructs like this. Surely we can do better. OK, how about a very literal, straight loop approach, just like we'd write in C?

```
mx=max(inds)
vec2=fltarr(mx+1)
for j=0L,n_elements(data)-1L do $
  vec2[inds[j]]=vec2[inds[j]]+data[j]
```

Accumulate based on the index. Not too bad. Runs much faster than using where(), but its run-time scales with the number of elements of data, independent to how many repeated indices there are.

Of course, anyone familiar at all with histogram() would realize there's a better route when many indices are repeated:

```
mx=max(inds)
vec3=fltarr(mx+1)
h=histogram(inds,reverse_indices=ri,OMIN=om)
for j=0L,n_elements(h)-1 do if ri[j+1] gt ri[j] then $
  vec3[j+om]=total(data[ri[ri[j]:ri[j+1]-1]])
```

This taps into the ever-so useful reverse indices vector to pick out those elements of data which fall in each "bin" of the index histogram. Notice I'm using OMIN to save time in case the minimum index is greater than 0. This is much faster than the where() method, and can be a factor of 2 or 3 faster than the literal loop approach, if indices are repeated at least a few times on average (a few drops in each histogram bin). If indices are never repeated, or especially if many indices are skipped (a *sparse* set), the literal loop method can be much faster than histogram.

This is all well and good, but Carsten was amazed that I'd offered a *loop* in a solution. I wondered whether a loop-free and possibly faster version existed.

Given the initial set of reverse indices, the problem reduces to one of: is there a loop-free way to decimate an array using a variable width "chunk" decimation? E.g.: total the first 5, then the next 3, then the next 0, then the next 7, ..., elements of an array. I was encouraged by the histogram(total(/CUMULATIVE)) method which solved Pavel's chunk fill problem of years past, and came up with:

```
mx=max(inds)
h1=histogram(inds,OMIN=om,REVERSE_INDICES=ri1)
col=ri1[n_elements(h1)+1:*]
 h2=histogram(total(h1,/cumulative)-1,MIN=0,reverse_indices=r i2)
row=ri2[0:n_elements(h2)-1]-ri2[0]+om ; chunk indices = row number
sparse_array=sprsin(col,row,replicate(1.,n_ind),(mx+1)>n_ind)
if mx ge n_ind then $
  vec4=sprsax(sparse_array,[data,replicate(0.,mx+1-n_ind)]) $
else vec4=(sprsax(sparse_array,data))[0:mx]
```

I use sparse arrays to solve the chunk decimation problem, with the chunk fill method generating the row numbers of non-zero (unity actually) entries, and the original reverse indices generating the column numbers. Unfortunately, RSI's Numerical Recipes-based sparse array routines demand square arrays (which seems unnecessary to me), so you either have to pad the data or truncate the result, depending on whether you have more repeated indices than skipped indices. Even

so, this method is at least 10 times faster than the former histogram() method for relatively dense (many repeated index) mappings! For sparse sets of indices, it still works (thanks to that if statement at the end), and, amazingly, can still beat the literal loop method! Such is the penalty for looping at all using IDL variables, that you're better off going to elaborate lengths creating sparse array structures and histograms just to get all your looping to occur in real compiled code.

For a random list of 20,000 indices drawn from 0-2000 (a dense sampling: each index repeated 10 times, on average), the methods time to (average, sec):

```
0.48   ; where()
0.024  ; literal loop
0.011  ; histogram() loop
0.0096 ; sparse array method
```

And for 20,000 indices drawn from 0-40,000 (a sparse sampling: only 1 out of 2 indices present on average), you get:

```
5.839  ; where()
0.0257 ; literal loop
0.1047 ; histogram() loop
0.0207 ; sparse array method
```

Yes, it's ugly, but the numbers speak for themselves. For those of you not too squeamish to look closely, you'll see that I've used a histogram of a histogram.

Does anyone begin to feel like the looping penalty in IDL is a bit much? In any case, it looks like I'm going to add the spr* functions to my rebin/reform/histogram/## power tool list. They seem to be quite fast.

JD
