Subject: REDUCE
Posted by John-David T. Smith on Fri, 30 Mar 2001 00:18:09 GMT
View Forum Message <> Reply to Message

IDL Monkeys:

If you've done any C+IDL programming, skip to the end for a (possibly impossible) programming challenge: WARNING: not for the faint of heart.

The little n-dimensional histogram thought exercise revved my juices to waste more time and finish a little project which had been languishing for months: REDUCE. For those of you without idelible memories of all things idl-pvwave, I was lamenting the lack of any good, multipurpose, built-in threading tool, i.e. something for applying operations over a given dimension of a multi-dimensional array, similar to the way "total" allows you to specify a total'ing dimension. Craig has also supplied us with cmapply, which, while useful, is forced to compromise speed.

The result is a C-program for building as a DLM and linking with IDL. Currently, the options supported are:

Operations:
"MAX"
"MEAN"
"MEDIAN"
"MIN"
"MULTIPLY"
"TOTAL"

Options:
"DOUBLE" - work in double precision
"EVEN" - for median, same as keyword in IDL's median() function

That is, you can take the median over the third dimension of a 5D hypercube, and so on.

REDUCE works in any native numeric type, preserving type if possible. For certain operations, namely, TOTAL, MEAN, and MULTIPLY, it is always performed in FLOAT (or DOUBLE if passed or natively present in the input). This is to avoid overflow, and follows the example of IDL's total(). I may add a keyword "NATIVE" to force working in the native type, overflows be damned, which might be useful in some instances.

Things I like about REDUCE:

1. It doesn't screw with your type unless it has to. For instance, IDL's median() converts everything but byte to float first. Why? REDUCE respects your right to use ULON64's or what have you natively.

2. It's fast. Preliminary testing indicates its from 2-50 times as fast as the same operation expressed in IDL (as you'd expect). Especially true for multiplies, but everything sees a healthy speed-up. It even takes medians faster than median().

The thing I don't like about REDUCE:

 It is horribly ugly.

The reason it is horribly ugly is all those damn types. If you followed Ronn Kling's book, you'd know he recommends handling multiple types like:

```
switch(type){
 case IDL_TYP_INT: myvar=(short *) foo;  stuff1; stuff2; stuff3; break;
 case IDL_TYP_LONG: myvar=(int *) foo;  stuff1; stuff2; stuff3; break;
 case IDL_TYP_FLOAT: myvar=(float *) foo;  stuff1; stuff2; stuff3;
break;
 ...
 ...
}
```

That is, just replicate things over and over again for the various types. REDUCE works natively in 9 types. Luckily, I didn't have to copy everything over nine times as above, but in essence that's what I did. I just used a host of clever C pre-processor directives to indirect the type replication.

OK, no problem. But what happens is "stuff" is large. For example, finding a median takes about 75 lines of code with all the initialization etc. What's more you need a separate copy of the same code not just for the 9 types, but also for the cases in which you're possibly promoting to double, or float. A given piece of code can end up being replicated 18 times, with slight differences like:

```
float *p=IDL_MakeTempArray(IDL_TYP_FLOAT,...);
```

vs.

```
short *p=IDL_MakeTempArray(IDL_TYP_INT,...);
```

vs.

```
int *p=IDL_MakeTempArray(IDL_TYP_LONG,...);
```

and so on, ad infinitum.

For a couple lines of code, this isn't too bad, but when you're forced
to shoehorn a 70 line function into a macro just to replicate it 9 or 18
times with some very subtle change, it gets ugly, and bloated. My
nested loops which do the magic of threading the calculations occur 81
times in the code, after pre-processing!   Yuck.

If this is how IDL handles dealing with multiple types internally, well
that makes me very sad (and allows me to understand why their median
only deals with two types).

The question, to all you C-programmers: is there a better way?

In order to phrase the challenge more sensibly, consider a function that
will take the maximum of an array of data:

IDL_LONG maximum(data)

The catch is data will be of whatever numeric type the user likes (see
the list in external/export.h under the IDL directory for a list of
them).

First recognize that the code logic to compute the maximum will be the
same, both symbolically for all types (e.g. "if data[i]>max then
max=data[i]"), and for many types, in the compiled code itself. Can you
come up with a portable way to write and call maximum() which avoids any
of the repitition intrinsic in the straightword approach, that is, to
avoid compiling in the code like

"if data[i]>max..."

once for each type?

Thought I'd give it a shot. I'll release REDUCE to the masses once I
sort these issues out.

JD

Subject: Re: REDUCE
Posted by Craig Markwardt on Mon, 02 Apr 2001 20:17:13 GMT

View Forum Message <> Reply to Message

JD Smith <jdsmith@astro.cornell.edu> writes:
> The thing I don't like about REDUCE:
>
> It is horribly ugly.
>
> The reason it is horribly ugly is all those damn types. If you followed

> Ronn Kling's book, you'd know he recommends handling multiple types
> like:
>
> switch(type){
> case IDL_TYP_INT: myvar=(short *) foo; stuff1; stuff2; stuff3; break;
> case IDL_TYP_LONG: myvar=(int *) foo; stuff1; stuff2; stuff3; break;
> case IDL_TYP_FLOAT: myvar=(float *) foo; stuff1; stuff2; stuff3;
> break;
> ...
> ...
> }
>
> That is, just replicate things over and over again for the various
> types. REDUCE works natively in 9 types. Luckily, I didn't have to
> copy everything over nine times as above, but in essence that's what I
> did. I just used a host of clever C pre-processor directives to
> indirect the type replication.

[Sorry if some of this appeared on Friday; somehow I don't think it
got sent.]

Hey JD--

REDUCE looks very cool. I think this is exactly the kind of thing
we'd like in the core of IDL.

As for your question about ugliness, I think it is unavoidable. You
were hoping for some way, at *runtime* to do some polymorphic
operations on the data. Unless you want to get into self-modifying
code, I think it's impossible. To see this, you only have to realize
that there are different machine language opcodes for different data
types such as float, int32, byte, and so on.

Since you want you operation to occur on any data type, at some
fundamental level all of these opcodes have to be exercised, thus it
has to occur at compile time, not runtime. You are doing it now with
macros. You could presumably do it using templates in C++. Them's
the breaks.

To go slightly off the main topic, you did mention the code to handle
the casting of data from one type to the other. This is quite
difficult to get right using the C interface to the IDL runtime. This
is actually a classic case where using the IDL language as a *wrapper*
is good. This is really quite straightforward: you do all the type
checking, casting, and other checking in an IDL script, which then
passes the resulting data to an internal routine. The internal
routine is thus freed from performing lots of complicated checking
itself and can get down to the business of crunching the numbers.

WRITE_GIF is a good example of this, despite the internal routine
ENCODE_GIF being disabled because of the GIF licensing malarkey.

While this doesn't answer your fundamental question, it may help to
remove some of the complexity from the C code.

Craig

--
 -- --------------
Craig B. Markwardt, Ph.D.   EMAIL: craigmnet@cow.physics.wisc.edu
Astrophysics, IDL, Finance, Derivatives | Remove "net" for better response
 -- --------------