Subject: Re: IDL as programming language?
Posted by steinhh on Sun, 28 Aug 1994 11:14:05 GMT
View Forum Message <> Reply to Message

These kinds of questions seem to recur quite often, so I'm posting instead
of just replying by E-mail.

In article <1994Aug28.122441.32497@waikato.ac.nz>, abz@waikato.ac.nz writes:
|>[...]
|> We have a number of questions, concerning
|> comparisons between IDL and other programming languages (particularly FORTRAN).
|> We are currently running an older version of IDL (2.2.2) on a Sun SPARC
|> station.
|>
|> (i) Accuracy. Our current version of IDL seems to prefer doing calculations
|> in single precision, while we prefer double. Has this been improved in the
|> latest version? (e.g. in our current version, routines like LUDCMP work in
|> s.p., despite being passed d.p. arguments.)
|>

I have little experience with this particular problem, but if you already
have written routines in FORTRAN that does specific jobs in double precision
(I guess you have),the use of the CALL_EXTERNAL routine makes it quite easy
to use those routines whenever the built-ins are not sufficient.

A note of advice on CALL_EXTERNAL: Don't let the hopeless bureaucratic style
of the supplementary information in "call_external.doc" baffle you. Simply
use your favourite editor to delete everything that has nothing to do with
your machine, operating system, programming language etc, and then do the
same with the examples. It's amazing how easy it is.

|> (ii) Speed. Some of us Grads are running some really time consuming programs
|> (large arrays, large loops). How does IDL compare with (say) FORTRAN in
|> general, speedwise? (my impression is that it's pretty slow, but I could be
|> wrong...)

For exactly those scenarios (FORTRAN loops processing large arrays), IDL
should be at worst a very few percent slower. If you're doing FFT's, the
built-in routine in IDL outperforms the standard Coley-Tukey version
in Numerical Recipes. I had a problem using FFT's for calculation of
autocorrelation functions of real-valued data, so I could use the
shortcuts in Num. Recipes twice (starting with real-valued data, and
the inverse transformation is also with real-valued data)
when implementing it in C, but I was only able to improve speed by
about 5-10% after squeezing it all out. I had thought that just writing
a straightforward implementation (without shortcuts) would save time, but
I ended up using *more* time! And the FFT's in IDL are written for
arbitrary array lengths, not just 2^N elements.

|>
|> (iii) Memory. How does IDL's memory management compare? Again, some of our
|> programs (FORTRAN) have a tendency to gobblelarge chunks of memory (probably
|> bad programming, but still...)
|>

This depends somewhat on programming style. For calculation-intensive
applications, it shouldn't be much worse than FORTRAN, although if you
are *really* "good" at it, you can make IDL spend a *lot* of memory.
(But this would happen in most languages, anyway).

|> (iv) What is a large IDL code like to debug?
|>

Generally, I find it very easy to debug. It's much easier to get
an overview of the programs compared to FORTRAN or C versions due to
the fact that array processing usually takes just one line instead
of being an explicit loop. Also, the ability to stop at any point
in the code, plotting data whenever you like etc. is very nice.

I have also found that it's very comfortable to use IDL as a development
tool, testing the algorithms in a very interactive way using IDL, and then,
if speed is essential, it's always easy to implement the hard work from
the IDL code into FORTRAN or C subroutines via CALL_EXTERNAL, or, in
extreme cases, to write a stand-alone program based on the IDL code.
The amount of time saved in testing and developing the algorithms is
tremendous!

|> (v) How 'robust' is IDL as a programming language? We have a variety of
|> different programming styles here -- some prefer 'quick and dirty' programming,
|> others a more structured approach. Forgive my possible ignorance, but I have
|> the impression that IDL as a language is more suited to the 'quick and dirty'
|> approach. Is this true? Does IDL as a programming language have many
|> glitches or inconveniences from a mathematical programmers point of view?
|>

I'm not sure I understand what you mean by 'robust', but here goes: IDL
doesn't have explicit declarations of variables, so you could say that
it's somewhat "quick and dirty" in that respect. Also, calling a routine
with too many parameters are only detected at runtime, and if you're
using too few parameters, it's up to the routine itself to check if
everything's OK (parameter typing etc as well) -- otherwise you could
get "Undefined" messages from the subroutine without really being aware
that you just didn't give it that particular piece of information. This
isn't a serious problem, though. Compared to, say C, IDL is *very* robust,
without any type checking (it doesn't core dump, and on PC's, you don't
risk deleting your harddisk if something goes haywire!). Implementation of

parameter #/type checks etc is, however, just as easy(-ier)
as any explicit declaration thereof would be in other languages. Also, an
advantage is that you can write a single version of most functions, and
they work just fine whether you'd like to pass it a scalar integer or a
double precision array, returning scalars or arrays respectively.

|> Any info/advice would be much appreciated. The types of stuff we do
|> here are generally large numerical (finite difference) codes on 2D and 3D
|> grids.

That's what IDL's for :-)

Stein Vidar
(Yes, I *do* like IDL)
