Subject: Re: Speed and array operations
Posted by JD Smith on Thu, 21 Jun 2007 18:37:30 GMT
View Forum Message <> Reply to Message

On Thu, 21 Jun 2007 09:33:01 -0700, Conor wrote:

> I've often wondered about the speed of IDL compared to other languages.
> Now, it is my understanding that IDL isn't really a compiled language
> (please, correct me if I'm wrong). Although the IDL interpreter
> "compiles" and IDL program before you run it, apparently it isn't compiled
> all the way down to machine code. As such, I'm under the impression that
> natively, IDL is always going to be slower than a fully compiled language
> such as C or fortran. However, it seems to me that there might be case
> where IDL might still be quicker. Namely, array operations. My question
> is, is it possible for IDL to be faster than a fully compiled language?
> Are the array operations in IDL so well optimized that adding together
> gigantic arrays in IDL would actually be faster than the equivelent
> for-loop style methods you would have to use in fortran or C? Or are C
> and fortran always faster? Or do I completely misunderstand how these
> languages work?

IDL can be "almost as fast as" a native compiled language [1,2], when you are doing something which stays "inside C code" for a long time. I.e. if you are using a built-in IDL primitive (like HISTOGRAM!) on large data sets, you might approach the performance of the same algorithm coded in C. In this case, all IDL is really doing is packaging up the data, and handing it off to some compiled code (written by RSI) which operates on it. This is really the same as your own "native" compiled program would do. Of course, you'd have to write it yourself!

When it comes to non-native code, i.e. plain old IDL code itself, this is compiled down to the equivalent of byte-code, and interpreted by a block of code inside IDL which is itself running natively (likely compiled from C sources ITTVIS wrote). I.e. it is running "one step removed" from machine code. However, often in the course of interpreting some chunk of this byte-code (for lack of a better term), IDL will have occasion to dump data into and collect data out of natively-compiled "blocks" of code.

Such blocks exist for many things, like all basic array arithmetic, and many of the functions you find in the IDL reference guide (save the ones which are "written in IDL"). The larger the fraction of time a given IDL program spends inside these "native blocks", the faster it will operate. This statement is the essence of all the vectorization and HISTOGRAM IDL optimizations you see. To maximize performance, package up data into as large chunks as possible (within memory limitations) and then get that data quickly into a native code block,

spending as much time in native code as possible.

In exchange for the power, flexibility, and programming simplicity offered by the IDL language, you take a speed penalty. This is no different from any other interpreted, higher-level language. In fact, IDL manages to keep many simple things quite speedy, compared to the others.

Of course, IDL is designed to be portable among different OS's first and foremost, and to produce correct results as well. This means that it is not, in my experience, aggressively optimized. For instance it does not make use of extreme compiler optimizations.

There are two "escape clauses" where IDL may offer quite competitive or even superior performance, at least compared to a single-threaded "normal" C of FORTRAN code:

[1] On a large multi-core system, the IDL thread pool allows it to easily spread simple calculations among many cores. There is a setup penalty to do this, but it can truly speed up simple operations on large data sets (array arithmetic, etc.). Of course, you could implement this threading in your C or FORTRAN version as well, and likely outperform IDL by a wide margin, but multi-threaded programming is not so easy, whereas IDL makes this trivial and "invisible".

[2] A very few IDL operations make use of the SIMD processor you likely have hiding in your CPU. There was a large fuss about the Altivec-awareness of IDL a few years back, around the time they canceled and then reincarnated IDL for Mac. Interestingly, I never heard much about IDL using SSE on Intel chips. The new SSE4 is (finally) very competitive with IBM's old Altivec unit. If (and it's a big if) IDL makes use of these units for certain operations, they could outperform C or FORTRAN implementations which did not do so. Again, if you coded to the SIMD unit yourself, you could realize these gains, likely many times over.

In practice, these cases are rarely encountered. I estimate a typical "good" non-trivial IDL algorithm is roughly 5-40x slower than if implemented in C. That's why I'm always amazed at these cluster-computing IDL solutions: if speed is so important that you want to throw a cluster at the problem, you would be better served implementing the most costly portions of your algorithm in C as a DLM, or as a separate process IDL communicates with. In the end though, it's a trade-off: developer time vs. run time.

JD
