SIGN IN

SCIENCE —

Scientific computing’s future: Can any coding language top a 1950s behemoth?

Cutting-edge research still universally involves Fortran; a trio of challengers wants in.

LEE PHILLIPS - 5/8/2014, 6:30 AM

Clojure—Lisp reborn

Lisp is experiencing a rebirth in the form of Clojure, which attains portability and access to a multitude of libraries through its implementation on top of the Java Virtual Machine. Clojure is inspiring a wave of developers who have always been curious about Lisp to finally give it a serious try. This is probably due in large part to a series of talks of inspiring clarity, available on YouTube and elsewhere, by Clojure’s author, Rich Hickey.

In addition to its JVM dependence (although there are implementations on other platforms, including JavaScript), Clojure adds many new ideas to the Lisp tradition, including new datatypes and several facilities to grapple with concurrency. Clojure was designed, through the use of “software transactional memory” and immutable data structures, to be a platform that makes it easy to reason about concurrent programming.

The Clojure logo.

ClojureTV on YouTube

Clojure's Rich Hickey.

Julia—good intentions

Julia is a very new language. It became ready to use for real work in early 2012. Nevertheless, it's already generated an excited buzz in scientific computing circles. Julia may be the first language since Fortran created specifically with scientific number crunching in mind. Julia allows expressive programming using sophisticated abstractions while attaining C (but not yet quite Fortran) speed in many benchmarks.

If Julia has a central idea, it is *multiple dispatch*: a function's behavior can depend on the types of (all of) its arguments. Julia has powerful concurrency and networked programming facilities; it can interface seamlessly with Fortran and C library routines; it allows Lisp-style true macros with conventional mathematical syntax; and it's able to act as shell-like glue code. Julia can be as simple and direct to program in as Python while offering an order of magnitude increase in speed.

The Julia logo.

Here is a version of the Fibonacci function in Julia:

```
function fib(n)
  if n < 2
     return n
  else
     return fib(n-1) + fib(n-2)
  end
end
```

This is very similar to the equivalent Python code, but it runs about 10 times as fast (and, as with Python, it will run out of stack space for moderately large values of `n`, as neither language offers tail call optimization. However, this might be in Julia's future).

Each of these languages, crucially, is free and open source. Each one can be operated from a REPL, an interactive prompt that allows programs to be built up in an exploratory fashion impossible with Fortran. This also allows the languages to be used as scientific calculators on steroids. Finally, from personal experience, there were no obstacles to downloading and installing all of these languages on a Linux laptop. Each allows you to start working immediately, and that's only helped by each language's friendly and vibrant online communities that come with excellent online tutorials and documentation.

Long live the king!

In the face of all this exciting progress in language design, it's instructive to examine the reasons for Fortran's continued leadership in technical computing in order to ask whether this is likely to continue.

Fortran emerged from IBM in New York City in the 1950s. The Fortran 1 compiler [PDF], according to author Maryna Karniychuk, was the "first demonstration that it is possible to automatically generate quality machine code from high-level descriptions." It was the first successful high-level language, and the compiler held the record for optimizing code for 20 years.

What computers looked like when Fortran was introduced.

Fortran has been consistently regarded as the fastest language available for numerical work, and it remains the standard used for comparatively benchmarking supercomputers. But what does it mean for a language to be fast?

Speed has little to do with the language itself and very much to do with the compiler. A dramatic and illustrative example is Java, which shed its reputation as a slow language with the development of Just-In-Time optimizing compilers. Java is now considered in the C-class, or nearly so, and it's routinely used where speed is important.

One of the reasons for Fortran's speed is the optimization made possible by the enforcement of strict aliasing. Although this was added as an option to the C99 compiler, classic C cannot make many of the optimizations available to the Fortran compiler because it must assume that, for example, arrays with different names might overlap in memory.

Another reason for the superlative speed of Fortran compilers is the decades of experience in creating optimized machine code for particular processor families. While generic, open-source Fortran compilers are available for a wide range of platforms and are of good quality, often the best performance can be had from a commercial compiler produced by a hardware manufacturer. For example, the Intel Fortran compiler can generate vector instructions and even message passing calls for optimized, distributed memory multiprocessing. The manufacturer, with intimate knowledge of the CPU, can ensure that the compiler produces machine code tuned to get the best possible performance from its architecture. Since supercomputers composed of their processing units will be compared with competitors using benchmarks written in Fortran, there is an economic and prestige-based motivation to invest in this level of compiler research and development.

It certainly seems likely, in light of the above, that Fortran will remain the fastest option for numerical supercomputing for the foreseeable future—at least if "fast" refers to the raw speed of compiled code.

But there are other reasons for Fortran's staying power. The profusion of legacy code and numerical libraries written in Fortran creates a powerful incentive for new projects to stick with the venerable language. There is an easy interface with excellent routines for linear algebra, special functions, equation solving, etc. There's also a good chance that one will need to incorporate or modify a colleague's code in your problem domain, and this code is likely to be in Fortran.

The popularity of Fortran is also supported by the attention to backward compatibility as the language evolves: F90 completely supports F77, although certain quaint language features from old Fortran are removed from more recent standards. These are such abominations as the "assigned GOTO" statement and the ability to GOTO an ENDIF statement from *outside* the IF block. ("Real Programmers like Arithmetic IF statements—they make the code more interesting.") This kind of thing encouraged generations of engineers and scientists from the '50s to the '90s to create impenetrable spaghetti code.

Another secret to Fortran's endurance is the evolution of the language standard. Fortran has chosen features from other languages conservatively, maintaining its status as a practical numericist's toolbox while remaining conceptually simple.

F90 introduced an option to dispense with the annoying rigid line format of earlier standards among its many other improvements. Best of all was array syntax, reminiscent of array-based languages such as APL. No longer were we required to write verbose and tedious loops over arrays, but users could instead operate on them as units: C = A + B was now allowed when A, B, and C were not just numbers but arrays of any rank, instructing the machine to add A and B elementwise and store the result in the corresponding location in C. This not only made code more succinct and pleasant to write, but it could give another hint to the compiler that a potentially optimizable array operation was going on. In Fortran 77, the array sum would have to be written this way:

```
      DO 100 J = 1, N
         DO 200 I = 1, M
            C(I, J) = A(I, J) + B(I, J)
200      END DO
100   END DO
```

Other major F90 improvements, such as modules and allocatable arrays, aided code reuse.

The most recent Fortran standard, Fortran 2008, introduced another major innovation: coarrays. This allows arrays to be split between processors for parallel processing, and it includes a natural syntax for interprocessor communication. With coarrays, parallel computation becomes part of the language specification rather than requiring a clumsy interface to an external library (although under the hood this is all actually implemented using MPI).

Unfortunately, compiler support is still limited. The most common open source compiler, gfortran, does not yet support coarrays, although work toward this is ongoing. However, the open source G95 compiler, under development, does support much of Fortran 2008, including coarrays.

Page: 1 2 3 Next →

READER COMMENTS 345

SHARE THIS STORY

← PREVIOUS STORY

NEXT STORY →

Related Stories

Today on Ars

RSS FEEDS
VIEW MOBILE SITE
VISIT ARS TECHNICA UK
ABOUT US
CONTACT US
STAFF
ADVERTISE WITH US
REPRINTS