SCIENTIFIC METHOD / SCIENCE & EXPLORATION

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.

by Lee Phillips May 7 2014, 9:00pm EDT

DEVELOPMENT OPEN SOURCE SUPERCOMPUTING 325

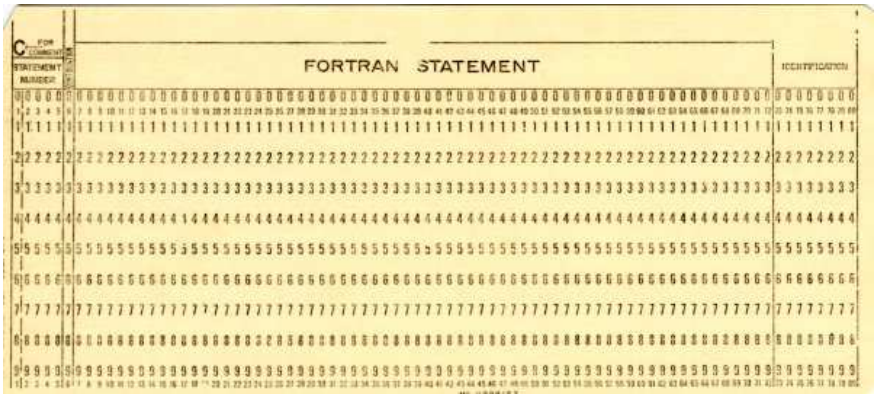
> *"I don't know what the language of the year 2000 will look like, but I know it will be called Fortran." —Tony Hoare, winner of the 1980 Turing Award, in 1982.*

Take a tour through the research laboratories at any university physics department or national lab, and much of what you will see defines "cutting edge." "Research," after all, means seeing what has never been seen before—looking deeper, measuring more precisely, thinking about problems in new ways.

A large research project in the physical sciences usually involves experimenters, theorists, and people carrying out calculations with computers. There are computers and terminals everywhere. Some of the people hunched over these screens are writing papers, some are analyzing data, and some are working on simulations. These simulations are also quite often on the cutting edge, pushing the world's fastest supercomputers, with their thousands of networked processors, to the limit. But almost universally, the language in which these simulation codes are written is Fortran, a relic from the 1950s.

Wherever you see giant simulations of the type that run for days on the world's most massive supercomputers, you are likely to see Fortran code. Some examples are atmospheric modeling and weather prediction carried out by the National Center for Atmospheric Research; classified nuclear weapons and laser fusion codes at Los Alamos and Lawrence Livermore National Labs; NASA models of global climate change; and an international consortium of Quantum Chromodynamics researchers, calculating the behavior of quarks, the constituents of protons and neutrons. These projects are just a few random examples from a large computational universe, but all use some version of Fortran as the main language.

This state of affairs seems paradoxical. Why, in a temple of modernity employing research instruments at the bleeding edge of technology, does a language from the very earliest days of the electronic computer continue to dominate? When Fortran was created, our ancestors were required to enter their programs by punching holes in cardboard rectangles: one statement per card, with a tall stack of these constituting the code. There was no vim or emacs. If you made a typo, you had to punch a new card and give the stack to the computer operator again. Your output came to you on a heavy pile of paper. The computers themselves, about as powerful as today's smartphones, were giant installations that required entire buildings. (OK, these computers only had *a fraction* of the power of today's smartphones.)



A Hollerith card that, when punched, will contain one Fortran statement.

Public domain

In the 60 years since the creation of the first Fortran compiler, there has been tremendous activity in the field of programming languages and computer science. Entire paradigms of language design and program organization have arisen and done battle with each other. Fortran has remained serenely detached, now and then incorporating an idea or two into a new version of the language.

While structured programming, object orientation, functional programming, and logic programming all arose to solve various problems that supposedly were not solved by primitive Fortran, none of the languages that embodied these new organizing principles came close to supplanting Fortran in the realm for which it was invented: scientific and numerical computing. This remains true up to the present, as shown by the examples above and by the content of courses and textbooks on the subject. *Introduction to High Performance Computing for Scientists and Engineers*, for example (published in 2010), contains most of its code samples in Fortran.

Now, a few years after the appearance of Fortran, a very different kind of language was invented: Lisp, or, as it was originally called, LISP. "Originally specified in 1958, Lisp is the second-oldest high-level programming language in widespread use today; only Fortran is older," writes Dr. Joey Paquet in his lecture notes for the Comparative Studies of Programming Languages class [PDF] at Concordia University. Although Lisp became popular with artificial intelligence researchers, it never caught on with physical scientists. Two main reasons for this are speed and weirdness. Speed, because although some versions of Lisp attained respectable runtime efficiency, they were not in the same class as Fortran for heavy numerical work. Weirdness, because the prefix notation used by Lisp made expressions in that language look a lot less like normal mathematics than did math rendered in Fortran. (Fortran stood, after all, for FORmula TRANslator.) A normal chemist or engineer is far more comfortable with **y = (a + b)/c** than with **(setf y ((/ (+ a b) c)))**.

This was, in a sense, unfortunate. The abstract power of the Lisp family, which is closely connected with its peculiar notation and its functional nature, can provide elegant solutions to problems of parallelism that Fortran eventually grappled with using a clumsy agglomeration of ad-hoc libraries and compiler directives hiding in code comments. But, as we'll cover, Lisp has been reborn in a modern form that is beginning to attract some numericists due to its powerful approach to concurrency.

Including the modern iteration of Lisp, three recently developed languages may actually have a chance to step out from the vast shadow of Fortran. Each hopes to capture the hearts of different segments of the scientific computing community.

Meet the candidates

Computer science and computing languages have been highly active fields of research since before Fortran Zero was unleashed. In that time, scores of languages have been designed and put to use, many of which provide powerful abstractions and facilities to the programmer that have little chance of ever making it into any version of Fortran. This is because many of the key ideas of these other languages are incommensurate with the imperative, data mutating design implicit in the core of Fortran. And these ideas are not merely academic curiosities; they afford real, practical, and provable advantages to the implementer of complex numerical algorithms that need to run correctly and efficiently in a multi-processor computing environment.

Haskell—the elder statesman

Of the three young languages with the potential to move beyond Fortran, the oldest is called Haskell. The unique programming language attained its first "stable variant" in 1998. While C, Fortran, and Pascal are part of the Turing machine branch of the language world, Haskell, along with Lisp, is a member of the lambda calculus branch. These programs are conceived as the composition of

functions rather than as a series of explicit steps.

It's difficult to convey the feel of programming in Haskell; one really needs to learn a bit of the language and try it out. And it's worth doing this for the unique experience of an entirely unfamiliar style of programming, even if you never actually use the language.

While Fortran provides a comfortable translation of mathematical formulas, Haskell code begins to resemble mathematics itself. With its elaborate type system and pattern matching, the beginnings of function definitions in Haskell look like the setups of proofs in mathematical monographs, with definitions and axioms carefully established.

A simple example of this is a naive definition for a function that returns the Fibonacci sequence (recall that the Fibonacci sequence can be defined recursively as $F_0 = 1$; $F_1 = 1$; $F_N = F_{N-1} + F_{N-2}$ i.e., each term is the sum of the previous two terms):

```
fib :: Integer -> Integer
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

The Haskell logo.

This may look like a definition extracted from a mathematics textbook, but it's legal Haskell code.

Haskell excels in situations where program *correctness* needs to be ensured, if possible before run time. Its purely functional nature allows programs to be constructed with a high degree of confidence in their correctness. In fact, it is a common experience for Haskell programmers to find that their code runs correctly the first time after all the compiler errors have been dealt with. This is a rare experience for users of almost any other language, and it's due to Haskell's sophisticated type system.

PAGE: 1 2 3 NEXT →

READER COMMENTS 325

← OLDER STORY | NEWER STORY →

YOU MAY ALSO LIKE

SCIENTIFIC METHOD / SCIENCE & EXPLORATION

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.

by Lee Phillips May 7 2014, 9:00pm EDT

DEVELOPMENT OPEN SOURCE SUPERCOMPUTING 325

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.

Clojure's Rich Hickey.
ClojureTV on YouTube

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.
NASA

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 325

← OLDER STORY | NEWER STORY →

YOU MAY ALSO LIKE

SCIENTIFIC METHOD / SCIENCE & EXPLORATION

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.

by Lee Phillips May 7 2014, 9:00pm EDT

DEVELOPMENT OPEN SOURCE SUPERCOMPUTING 325

Contenders or pretenders to the throne?

Considering all the above reasons for Fortran to stay dominant, what are the chances that another language can carve out a niche in this realm?

Haskell, Clojure, and Julia each provide features and abstractions that Fortran, despite its willingness to incorporate new ideas in its evolving standards, will probably never embrace. Some of these characteristics are simply too alien to the Fortran tradition, and others are actually incommensurate.

The world of large-scale scientific computing carries with it punishing constraints on performance. If you're calculating something that takes five minutes, you might consider living with a fourfold increase in run time if it means you can use a fun and expressive language. But if your calculation takes a week using Fortran, you probably will not be willing to wait a month, even if you get to use clever macros in your code.

All that is to say that while other languages might offer strong increases in programmer productivity or powerful abstractions that allow the researcher to envision new approaches to a solution, they stand no chance in the supercomputing world unless they get within a factor of two in runtime of optimized Fortran code. So how do our chosen languages fare?

Still suffering from the need for speed

Clojure was designed from the ground up with the problems of concurrent processing in mind. Software transactional memory, the functional paradigm, and immutable data can make all the problems of contention over memory go away, making it easier to reason about concurrency.

The peculiarly expressive nature of Clojure, which it shares in some form with all Lisps, allows beautifully compact representations of mathematical and computational ideas. Here is a definition of an infinite array that contains the entire Fibonacci sequence:

```
(def fibs (lazy-cat [0N 1N] (map + fibs (rest fibs))))
```

When you want a particular Fibonacci number, you can pluck one out of this by saying, for example:

```
(nth fibs 100)
```

This particular implementation of the Fibonacci sequence is not ideal in that it uses a lot of memory, but it's a neat example of a concept called *corecursion*, where you can build up an infinite data structure from the inside out.

Prof. Lee Spector, director of the Hampshire College Computational Intelligence Laboratory, described in an e-mail why he uses Clojure in his work:

> I do expect functional languages to continue to grow in popularity. For me, the main benefits stem from the high-level abstraction-building tools that functional languages tend to provide. I would and do indeed develop simulations in Clojure, rather than languages like C/C++, because the abstractions that it provides allow me to turn ideas into code and modify code as my ideas change, much more quickly.

But does Clojure make concurrent programming easier? Spector replied:

> I do also think that Clojure makes the expression of parallel (multicore) computations relatively straightforward, and that is also part of the reason that I use it. The combination of immutability and well-thought-out primitives for thread and memory control can make it simpler to design and implement parallel algorithms. Actually getting the hoped-for multicore speedups, however, can be more difficult[.]

In his last comment, Spector is referring to problems that he and others had in getting Clojure code to fully utilize all the processing units available to it. This is still an unresolved problem, but it may have to do with the JVM rather than the Clojure language itself. Whatever its origin, until it is resolved, this language, which manages to be both practical and beautiful, will not find a place among the most demanding scientific and engineering applications of supercomputers.

Moving on, at least one computational physics course is taught using Haskell. Haskell has also been used in a variety of Monte-Carlo physics simulations, which benefited from the relative ease with which the language allows one to reason about concurrency. Haskell has also been used to generate C code for Monte Carlo chemistry calculations. Its most common implementation using the ghc compiler generally yields good numerical performance on single processors.

However, there are several obstacles to Haskell's potential adoption as a numerical workhorse. Although Haskell and ghc support several forms of concurrency and parallelism, this is still under development, and it remains to be seen whether this can be pushed to the performance levels that would put it on the Fortran radar.

The extreme *differentness* of Haskell, although affording many advantages in an absolute sense, means that its adoption requires a period of deep study and immersion. No doubt there are scientists who would be glad to undertake this, but very few are willing to spare the time if there are doubts about speed. Another obstacle to performance is the lack of motivation of CPU manufacturers to work on something as exotic as a Haskell compiler. Consequently it is unclear if ghc will be able to generate vectorization code, for example, for processors of interest.

An important part of Haskell's design is "laziness;" this means that a calculation is not actually performed until its result is needed. This aids expressiveness, allowing you, for example, to use what are formally operations on infinite series, but ultimately using only a finite subset. (Clojure has lazy sequences as well, but laziness can be avoided there while remaining idiomatic.)

Haskell's laziness makes it hard to predict the time and space utilization of a routine. Haskell's nature makes it unusually easy to reason about a code's correctness, but it's harder to reason about its computational complexity. A Haskell code's time and space behavior may change for differently sized data. You can enforce strict (non-lazy) behavior, but that damages much of the expressivity of the language.

Haskell and Clojure are examples of functional languages, where computations are expressed, as much as possible, as compositions of *pure functions* that are routines that return a result and create no side effects. An important aspect of these languages is that data is *immutable*: data structures are transformed by functions by making altered (logical) copies and are never changed in place.

The aliasing issue that is a large factor in making Fortran faster than classic C becomes irrelevant when using a pure functional language. Since data is immutable and there is no writing to memory, the entire issue becomes moot. Entire classes of hard-to-find bugs become *impossible to create*.

Immutable data makes it easier to reason about code and to write correct concurrent programs. However, these very advantages lead to problems when trying to do computational physics in the large. Fast, large-scale numerical code depends on destructive updating of large arrays. This goes against the grain of the functional approach. Haskell and Clojure can be coerced to hack on memory locations, but at the cost of sacrificing many of their natural idioms and even violating their philosophies. Hickey, the author of Clojure, emphasizes the distinction between *place* and *value*, suggesting that computation should be thought of as the transformation of the latter. But fast numerics would seem to require dealing with *place* in the sense of locations in arrays.

Thus we face a rather fundamental conflict between the legacy of numerical algorithms and some of the most valuable developments in language design. This is another factor that helps Fortran remain the language of choice for numerical scientific work, because most of our familiar numerical algorithms have no impedance mismatch with Fortran's mutable-array paradigm.

Formidably fast

In contrast to the two functional languages above, **Julia** makes a big dent in the legacy library issue. Fortran and C routines (as long as they live in a shared library) can be called directly, even from the REPL, with no glue code and no ceremony.

Julia's published benchmarks show it performing close to or slightly worse than C, and Fortran, as usual, performing better than C for most tasks. These are single-processor benchmarks, however, and it remains to be demonstrated whether Julia can give good parallel speedups in large-scale numerical problems. Its support for concurrency and parallelism is deep and varied, however. The team that created and is developing Julia—namely Stefan Karpinski, Viral Shah, Jeff Bezanson, and Alan Edelman—were kind enough to respond to some questions by e-mail. On the subject of concurrency, they say:

> Julia makes it easy to connect to a bunch of machines—collocated or not, physical or virtual—and start doing distributed computing without any hassle. You can add and remove machines in the middle of jobs, and Julia knows how to serialize and deserialize your data without you having to tell it. References to data that lives on another machine are a first-class citizen in Julia like functions are first-class in functional languages. This is not the traditional HPC model for parallel computing but it isn't Hadoop either. It's somewhere in between. We believe that the traditional HPC and "Big Data" worlds are converging, and we're aiming Julia right at that convergence point.

The multi-paradigm nature of the language allows the natural expression of both functional and traditional numerical algorithms. But as we mentioned earlier, the central concept in Julia is multiple dispatch. According to the Julia team, this is the key distinction:

> Multiple dispatch is the language feature that really sets Julia ahead of the pack for numerical computing. Mathematical code tends to separate into high-level generic algorithms that could be applied to lots of different implementations of numeric types and structures on the one hand and very specialized custom implementation of those numeric types and operations on the other. Multiple dispatch is what lets you join these together smoothly—it lets you separate the usage of an abstraction from implementations of it, cleanly, in a way that we've never encountered in other systems.

And the problem of the lack of compilers for non-Fortran languages is at least partially solved by Julia's use of LLVM:

> "Big Iron" machines (think IBM, Cray) used in high-performance computing (HPC) circles often have only C or Fortran compilers. But with new projects like LLVM, it's increasingly possible to get good compiler support for many different languages on new hardware just by adding a new backend for LLVM—which is significantly easier than writing a C compiler from scratch. Recently, for example, someone got Julia running on ARM and we did nothing special to make that possible.

Finally, the Julia authors place a high priority on numerical performance. They told us that they have the runtime speeds at the top of their priorities list:

> Our general goal is not to be worse than 2x slower than C or Fortran and to allow programmers to easily trade off abstraction for more performance. When you write C-like code, you get C-like speed, but if you write functional style code with lots of higher-order functions, you may not get as much performance[.]

Although it's sporting a low version number and is under active development, Julia seems to be ready to use for real work today. Even if large scale computing projects are not your interest, the Julia REPL provides a very powerful and pleasant scientific calculator that understands rational and complex numbers, matrices, and a wide array of special functions.

The future

In the half century since the arrival of Fortran, the world of computing has undergone a series of quiet revolutions. We've left behind punch cards, 50 pound printouts on fanfold paper, and spinning reels of magnetic tape (mostly). We've also left behind the most irksome aspects of Fortran itself: the rigid line format, six-character variable names, static allocation, global common blocks following us everywhere, and much more. Fortran, beginning with F90, has the feel of a modern language. It continues to evolve and, with care and discrimination, adopt the most useful features from other languages. Lately we've seen the accretion of functional concepts onto Fortran with the PURE and ELEMENTAL keywords in F95. There is even a RECURSIVE keyword, which allows a function to call itself; something traditionally forbidden in Fortran land.

What does the future hold? Clearly, for one thing, the continued embrace of C++ for certain large

scientific computing projects. This has become possible in recent years due to advances in C++ compiler optimization and the ability to inform the compiler of the absence of aliasing. C++ has made significant inroads in the experimental particle physics community, where a commonly used suite of data analysis tools has actually evolved from a Fortran to a C++ version. The main reason for this seems to be that the complicated, interrelated structure of the massive pile of data from particle accelerators can be dealt with more conveniently using the pointer arithmetic and datatypes idiomatic to the C family.

And what about the newer languages discussed in the previous pages?

The Julia team offered these salient remarks in response to a question about the possible roles of Haskell and Clojure:

> When you have, for example, a huge array—one that barely fits in memory—you can't really afford to make any copies of it or represent it as anything besides a big contiguous chunk of mutable memory. For really hard computational problems, it is often the case that the only known tractable algorithms make extensive, essential use of mutable state. In a sense, Clojure and Haskell are tackling some of the hardest problems in computer science while Julia is aimed at the hardest problems in computational science. Someday, we may know how to deal with both at the same time, but for now, I think we have to choose our battles.

This may be a key insight that explains why highly sophisticated languages, the products of the brightest minds in computer science, fail to achieve impressive performance on tough numerical problems. Numerical performance at the large scale is a tough, distinct problem all its own, and only a language designed with this set of issues specifically in mind is likely to be able to grapple with it. The Lisp family, the object oriented paradigm descended from Smalltalk and the functionally pure languages such as Haskell all have their opinions about what they want to get right. But until the arrival of Julia, the only major language that was specifically designed to get large-scale numerics right was Fortran.

With this insight in place, it's not surprising that Fortran has no real competition (except perhaps for C++, but few people actually *enjoy* speaking that language) for serious numerical work. Clojure and Haskell can attain respectable performance on medium-scale projects, and, as pointed out by Prof. Spector above, their powerful abstractions make them preferable to lower-level languages wherever they can serve. But the most demanding numerical applications will always require a language designed with numerical algorithms in mind.

But once an ecosystem of libraries and tools develops around Julia, it stands an excellent chance of becoming an imposing presence in the scientific computing realm. Julia was intended for demanding numerical work, but it does not skimp on powerful abstract affordances, which helps immensely in the concise expression of algorithms and in concurrency. The epigraph that opens this article notwithstanding, there is a reasonable chance that the language of choice for scientific computing in another decade will be called "Julia."

PAGE: 1 2 3

READER COMMENTS 325

← OLDER STORY | NEWER STORY →

YOU MAY ALSO LIKE