DIRT SIMPLE HPC: MAKING THE CASE FOR JULIA

January 26, 2016 Douglas Eadline

Choosing a programming language for HPC used to be an easy task. Select an MPI version then pick Fortran or C/C++. Today there are more choices and options, but due to the growth of both multi-core processors and accelerators there is no general programming method that provides an "easy" portable way to write code for HPC (and there may never be). Indeed, it seems writing applications is getting more complicated because HPC projects often require a combination of messaging (MPI), threads (OpenMP), and accelerators (Cuda, OpenCL, OpenACC). In other words, to the HPC code practitioner things seem to be getting harder instead of easier.

This situation may be tolerable to the hardcore HPC applications developer, but it is a huge impediment to anyone wishing to get started in HPC. Consider the student who has a numerical problem to solve. An effective first strategy would be to use tools that can quickly test the feasibility of a solution. Several easy to program solutions might include, R, Matlab, Octave, or Mathematica. If the problem domain is small and the performance adequate, these packages are great solutions. They offer a level of programming abstraction that is often closer to the users application and further away from the details of the underlying hardware. In addition, they allow the user to easily tinker with the application and often include easy to use graphing and plotting capabilities. Though easy to use these applications often suffer a performance penalty when run in a production environment.

Consider the case where the student's new application needs to address a larger or more complicated data set or needs to examine a large number of data sets. Inefficiencies limitations that did not matter at first may now start dominating the application. The application may work, but other issues have also become important. Quick and interactive prototyping is usually associated with poor performance due to interpreted (not complied) code. Improving performance may be difficult for some applications and the only option may be to re-write the application in a more efficient language. In years past, users could count on the next generation processor (free lunch) to boost performance.

A second challenging issue is integration with other projects and libraries. This need often comes as applications grow out of the prototype stage and need to operate in a connected and multi-language computing world. Lack of or difficult integration can often lead to reinventing the software wheel (e.g. write your own solver).

Another issue is how to express parallelism in an application. At a minimum, applications should at least take advantage of multi-core processors in HPC environments because CPU cores are often there for the taking. In addition, methods on how to use using addition systems (cluster nodes) or GPU (accelerators) resources need to be factored in to the application code. Unfortunately, there is often a large impedance mismatch between the easy to use solutions and multi-core, GPU, and distributed computing methods used in HPC. In once sense, the need to insulate the new programmer from the complexity of these issues is essential. On the other hand, these capabilities need to be available to the programmer as they gain experience.

The above situation does not bode well for the neophyte HPC developer. Getting HPC into the hands of more domain specialist is not about access to better or faster hardware. Indeed, programming solutions that allow both quick victories and a growth path to enhanced HPC capabilities are needed.

MEET JULIA

Julia is a relatively new language, developed at MIT, that can help with some of the challenges mentioned above. The project team's mission is to create a free and open-source language that is general purpose and also excels at numerical computing and data science. This combination reduces the need for researchers to learn and use multiple programming languages to perform different computational analysis. Interestingly, many HPC practitioners are not aware of Julia or its capabilities. In one sense, Julia can be considered an HPC BASIC that is easy to use and at the same time allows users to tinker with advanced ideas and concepts. [Note: The author is well aware of Chapel (and others) and plans appropriate coverage in the future.]

The first thing to learn about Julia is that the developers are not bashful about their goals. According to the Why We Created Julia page:

*We want a language that's open source, with a liberal license. We want the speed of C with the dynamism of Ruby. We want a language that's homoiconic (i.e., has the same representation of code and data), with true macros like Lisp, but with obvious, familiar mathematical notation like MATLAB. We want something as usable for general programming as Python, as easy for statistics as R, as natural for string processing as Perl, as powerful for linear algebra as MATLAB, as good at gluing programs together as the shell. Something that is dirt simple to learn, yet keeps the most serious hackers happy. We want it interactive and we want it compiled. We want to write simple scalar loops that compile down to tight machine code using just the registers on a single CPU. We want to write A*B and launch a thousand computations on a thousand machines, calculating a vast matrix product together.*

The motivation and goals behind Julia often excite HPC users because many have first hand experience with the difficulties of writing good HPC software. Julia is maturing and sports a rather unique feature set:

- Syntax similar to MATLAB
- Designed for parallelism and distributed computation (multicore and cluster)
- Call Python functions: use the PyCall package
- Coroutines: lightweight "green" user-space threading
- C/Fortran functions called directly (no wrappers or special APIs needed)
- Powerful shell-like capabilities for managing other processes
- User-defined types are as fast and compact as built-ins
- LLVM-based, just-in-time (JIT) compiler that allows Julia to approach and often match the performance of C/C++
- An extensive mathematical function library (written in Julia)
- Integrated mature, best-of-breed C and Fortran libraries for linear algebra, random number generation, FFTs, and string processing
- Free and open source (MIT licensed)

The above list is by no means complete. There are many other features available in Julia. Perhaps the most impressive aspect of Julia is the "compiled C" performance that is available in an interactive environment. The following abbreviated table shows performance relative to compiled C (gcc). A larger version of the table with more languages is available on the Julia Page.

	FORTRAN	JULIA	PYTHON	R	MATLAB	OCTAVE	MATHEMATICA
Version	gcc 5.1.1	0.4.0	3.4.3	3.2.2	R2015b	4.0.0	10.2.0
fib	0.70	2.11	77.76	533.52	26.89	9324.35	118.53
parse_int	5.05	1.45	17.02	45.73	802.52	9581.44	15.02
quicksort	1.31	1.15	32.89	264.54	4.92	1866.01	43.23
mandel	0.81	0.79	15.32	53.16	7.58	451.81	5.13
pi_sum	1.00	1.00	21.99	9.56	1.00	299.31	1.69
rand_mat_stat	1.45	1.66	17.93	14.56	14.52	30.93	5.95
rand_mat_mul	3.48	1.02	1.14	1.57	1.12	1.12	1.30

Table One: Benchmark times relative to C (smaller is better, C performance = 1.0).

ADDRESSING AN HPC NEED

Based on performance alone, Julia provides a compelling development environment for HPC. There are, however, many other important features designed to support HPC application development. As an interpreted language Julia can provide an interactive experience to new users. Indeed, new users can even be spared the command line experience by using the iJulia Notebook that is similar to the popular iPython notebook. An example is shown in Figure One.

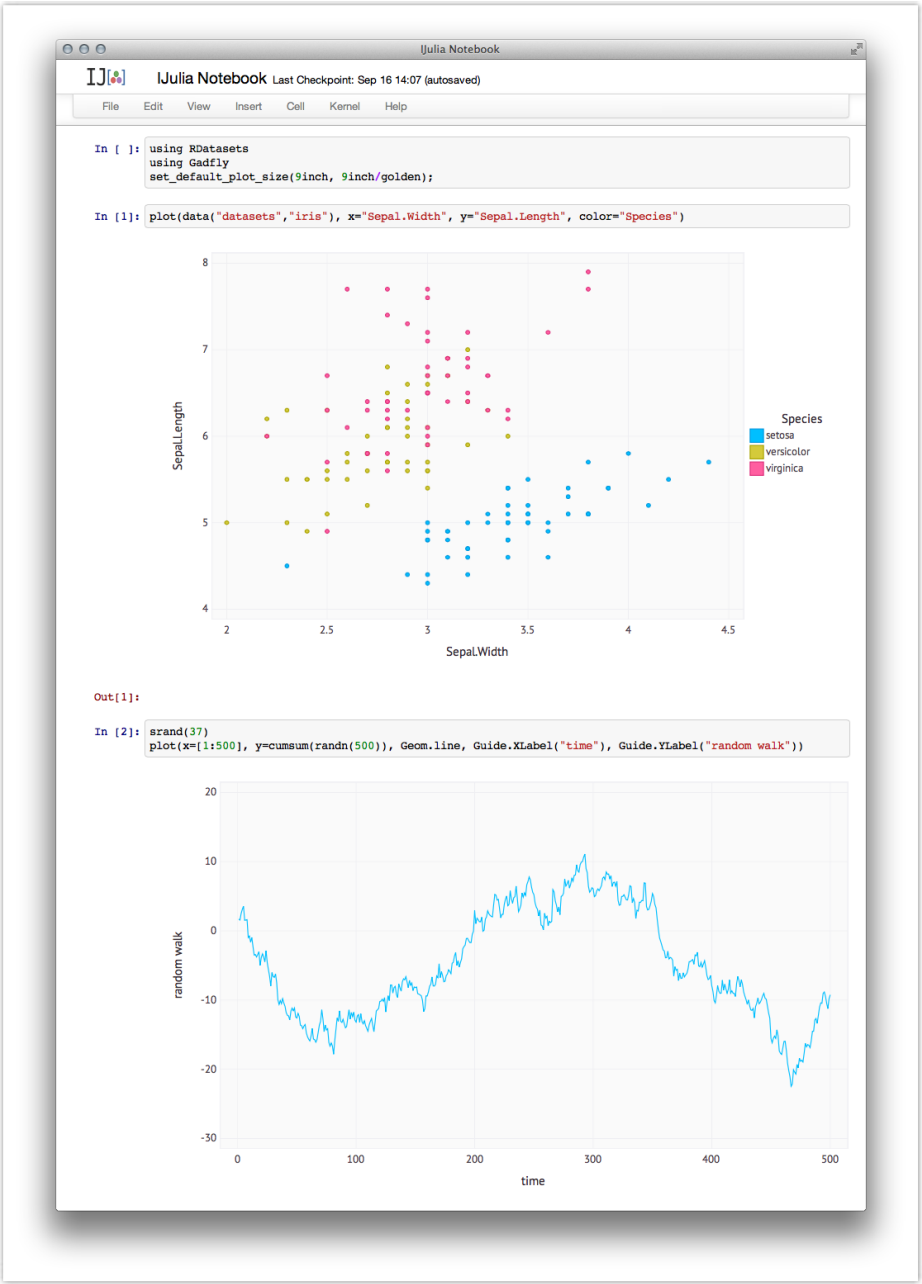


Figure One: The iJulia workbook GUI

As stated, Julia syntax is “Matlab like” and allows uncomplicated expression of mathematical formulas. For example a simple function to calculate the volume of a sphere can be written as:

function sphere_vol(r) return 4/3*pi*r^3 end

Using the interactive nature of Julia users can build-up and tinker with applications quickly. In addition, as indicated in Table One, users can also expect good performance “out of the box” for Julia applications due to the use of the LLVM-based just-in-time compiler approach.

To help integrate into existing work flows, Julia provides the ability to use both external programs and libraries from other sources. Rather than “shell-out” external programs Julia use forks/execs to launch and manage applications internally. Julia can use existing shared libraries compiled with gcc, gfortran, or Clang tools without any special glue code or compilation. This capability extends

to interactive sessions. The result is a high-performance, low-overhead method that lets Julia leverage existing libraries. In addition, existing Python code can be used by Julia though the PyCall package. This utility imports a Python module and provides Julia wrappers for all of the functions and constants with automatic conversion of types between Julia and Python.

An essential feature of Julia is native support for parallel programming. With Julia parallel computing is based on using multiple processes that can be run on either remote machines and/or local cores. Additional remote or local processes can be added at run time or by using a Julia cluster manager. Currently Julia supports SGE, Slurm, Scyld, HTCondor, and a local (same node) cluster managers.

The Julia parallel computing model sends messages between processes, but is different from MPI applications. Communication in Julia is generally "one-sided," meaning that the programmer needs to explicitly manage only one process in a two process operation. Julia parallel operations typically do not look like send/receive but rather resemble higher level operations like remote procedure calls to user functions. Similar to most other parallel programs, Julia processes are mapped to specific cores by the host operating system.

Parallel programming in Julia is built on two primitives: remote references and remote calls. A remote reference is an object that can be used from any process to refer to an object stored on a particular process. A remote call is a request by one process to call a certain function on certain arguments on another (possibly the same) process. Using these primitives, Julia provides support for parallel loops, distributed arrays (over multiple nodes), shared arrays (on a single node), channels, and synchronization. There is also support for custom message transports. Parallel programming features are optional and not required to start writing Julia programs.

Finally, there is support for GPU programing in Julia. Consult JuliaGPU for packages that support CUDA and OpenCL programming.

MEETING JULIA

Julia binaries are available for Windows, Macintosh, and Linux. As an open project, full source code is also available. Julia can also be run directly from a browser by connecting to JuliaBox (include an on-line tutorial).

Julia is still maturing and is fully capable of doing real work in the HPC community. The Julia ecosystem and community area growing quite rapidly and offer new libraries and applications. There are other interesting aspects of Julia that can be found on the project web site. The documentation is quite good and there are several tutorials available. Overall, Julia is a welcome addition the HPC community. Recently, the MIT based development team formed Julia Computing to provide support, training, and consulting services for the Julia language. The company has also received a two year $600,000 grant from the Gordon and Betty Moore Foundation's Data-Driven Discovery Initiative for further open source Julia development. The future looks bright for Julia. New and existing HPC coders will will appreciate a dirt-simple on-ramp to the HPC superhighway.

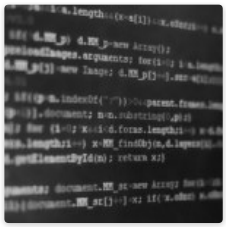
SIMILAR VEIN

Exascale Code Performance and Portability in the Tune of C

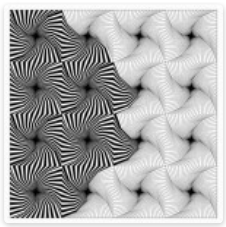
Amazon Gets Serious About GPU Compute On Clouds

AMD Lays Software Foundation For Renewed Hybrid Push

Programming Challenges on the Road to Summit's Peak



Inside The Programming Evolution of GPU Computing



Convergence Coming for Supercomputing, Machine Learning

Categories: Code, HPC

Tags: Chapel, GPU, Julia

The Bits And Bytes Of The Machine's Storage

Azure Stack Gives Microsoft Leverage Over AWS, Google

5 THOUGHTS ON "Dirt Simple HPC: Making the Case for Julia"

Cristian Vasile **says:** January 26, 2016 at 10:47 am

The guys behind SPARK, the crew who established DataBricks will integrate Julia in not so distant future. Will be interesting to see how HPC tasks and BigData tasks converge under Spark umbrella.

Reply

DA **says:** January 26, 2016 at 6:41 pm

Oh no, not another language please :–(, over my thirty years I've learned (and used) C, C++, Perl, Python, tcl, LISP (skill, scheme, Franz, common), FORTRAN, PROLOG, OPS5, BASIC, Matlab, Verilog, Mainsail, APL, javaScript, awk, shell, etc.

Just give me some decent encapsulation around shared memory, asymmetrical, multiprocessor/machine, etc. infrastructure/libraries for C++ and I'll be happy 🙂

Enjoying your articles…

Reply

OranjeeGeneral **says:** January 28, 2016 at 6:07 am

Sounds intriguing just too bad you need a google account to sign into JuliaBox to give it a quick spin :(. Have to install it probably instead

Reply

bob **says:** February 13, 2016 at 1:21 pm

"Consider the case where the student's new application needs to address a larger or more complicated data set or needs to examine a large number of data sets".
Create a boinc project!

Reply

juan says: July 19, 2017 at 3:35 pm

What are the advantages of using Chapel instead of other new parallel programing languages such as Liszt or Loci?

Reply

LEAVE A REPLY

Your email address will not be published. Required fields are marked *

Comment

Name *

Email *

Website

Post Comment

PAGES

About
Contact
Contributors
Newsletter

RECENT POSTS

Intel Stacks Up Xeons Against AMD Epyc Systems
Assessing The Tradeoffs Of NVM-Express Storage At Scale
Cavium Is Truly A Contender With One-Two Arm Server Punch

Medical Imaging Drives GPU Accelerated Deep Learning Developments

Mainstreaming HPC Codes Will Propel The Next GPU Wave