Newsletter:

Email »

Search:

Search »

News Features Blogs White Papers Archives Special Editions DigiSub Shop

Desktop Development Hardware Security Server Programming Operating Systems Software Networking Administration

Home » Issues » 2016 » 182 » Julia Login

Creating parallel applications with the Julia programming language

Getting Parallel

Share / Save

Article from Issue 182/2016

Author(s): Mark Vogelsberger

© Lead Image © Kavram, 123RF.com

Parallel processing is indispensable today – particularly in the field of natural sciences and engineering. Normal desktop users, however, can also benefit from higher performance through parallel execution with at least four calculation cores.

Programming tools such as MPI and OpenMP offer parallel processing features. It is easy to use these parallel language extensions, but using them efficiently is difficult because many algorithms cannot be rewritten for them. Languages such as Python and R also include parallel extensions, but these extensions were added on after the original language development and tend to be extremely slow when it comes to numerical calculations.

Many developers are looking for a language that is specifically designed with the intention of supporting parallel processing, and they want this parallel language to be easy to handle, with built-in features that facilitate parallelization and offer performance close to the blazing speed of C. A new language called Julia was developed to fill this niche (see the box titled "Julia Performance").

Julia Performance

Table 1 shows some of the micro benchmarks for different programming languages and problems published on the Julia website. The values are normalized so that C is equal to 1 and smaller values indicate better performance. Julia has almost C-like performance for some of the micro benchmarks and is much faster than languages like R and Python.

Other examples confirm this performance edge. For instance, you can use either Python or Julia to calculate many square roots in a loop. The examples explicitly write the loop from:

```
import math
import time
tstart = time.time()
for i in xrange(100000000):
 math.sqrt(i)
tstop = time.time()
print tstop-tstart
```

This loops needs about 10 seconds on the test system. The following Julia code executes exactly the same operations:

```
tstart = time()
for i = 1:100000000
 sqrt(i);
end
tstop = time()
println(tstop-tstart)
```

and takes less than 0.1 seconds. The Julia code is therefore more than 100 times faster than the equivalent Python code. Multiple dispatch with function calls gives Julia extremely efficient code that is practically superior to any high-level language. Faster code in Julia can be achieved without any tricks like vectorization or outsourcing to C extensions. By contrast, such tricks are often necessary to speed up Python or R code.

Table 1 Micro Benchmarks

	Fortran	Go	Java	JavaScript	Julia	MATLAB	Python	R
Version	Gcc 4.8.2	01.02.01	1.7.0_75	V8 3.14.5.9	0.3.7	R2014a	02.07.09	3.13
Fib	0.57	2.20	0.96	3.68	2.14	4258.12	95.45	528.85
Parse_int	4.67	6.09	5.43	2.29	1.57	1525.88	20.48	54.30
Quicksort	1.10	2.00	1.65	2.91	1.21	55.87	46.70	248.28
Mandel	0.87	0.71	0.68	1.86	0.87	60.09	18.83	58.97
Pi_sum	0.83	1.00	1.00	2.15	1.00	1.28	21.07	14.45
Rand_mat_stat	0.99	3.71	4.01	2.81	1.74	9.82	22.29	16.88
Rand_mat_mul	4.05	1.23	2.35	14.58	1.09	1.12	1.08	1.63

Hello Julia!

The current Julia version is available from GitHub [1]:

```
git clone https://github.com/JuliaLang/julia.git
```

Most Linux distributions also include Julia packages. Julia is a very young language that is constantly in development, so if you want the latest features, you should look for the latest developer release.

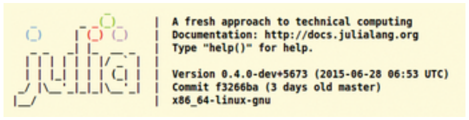
If you get the latest Julia source code from GitHub, you'll need to compile it using `make`. Compiling the code could take a few minutes – it should all run smoothly on most systems; however, the `README.md` file explains some cases where you need to manually readjust. After you compile, the `julia` binary files will appear in the root directory.

When you start it, you will see the Julia prompt shown in Figure 1. The dance starts when you enter a command at the Julia prompt. Programmers can try a Hello World example:

```
julia> for i=1:10
 println("Hello World ",i)
 end
```

produces the following output:

```
Hello World 1 Hello World 2 Hello World 3 ...
```



```
| A fresh approach to technical computing
| Documentation: http://docs.julialang.org
| Type "help()" for help.
|
| Version 0.4.0-dev+5673 (2015-06-28 06:53 UTC)
| Commit f3266ba (3 days old master)
| x86_64-linux-gnu
```

Figure 1: Julia's welcome screen right after the start.

Julia's syntax is heavily based on MATLAB. For example, indexing arrays starts at 1, not at 0. The syntax of for-loops is also the same as MATLAB. See the official Julia documentation [1] for more on the basic language structure.

Parallel with Julia

To parallelize loops using threading, programmers break up the loop and assign a part of the loop to each of the system's processors. The whole loop is processed faster because all the processors are computing in parallel. Another popular form of parallelization is distributed-memory parallelization, which involves calls for distributing both the computing work and the data.

Julia still uses another approach – a master-worker architecture or a one-sided-message passing. A process (the master) gives instructions to other processors (slaves). Programmers therefore only need to explicitly manage a process. The whole Julia parallelization is based on two main structures: remote references and remote calls.

A remote reference refers to an object that is connected to another process. For example, the result from a computation provided by another process can be used locally by using a reference. Similarly, a remote call is a call that allows a function to be executed by another process. Programmers can let any number of Julia functions be executed via remote call by any number of processes. All other parallel language structures in Julia use remote references and remote calls to deal with parallel tasks.

To operate Julia in parallel, users need to inform the program at the start how many processes should be started. If, for example, the machine has four processor cores, it could start four processes using the following:

```
./julia -p 4
```

You could also start more processes, although this wouldn't see any improvement in performance if you only had four physical cores.

You can add processes on the fly using the `addprocs()` function. Each process has an ID – the first master process is assigned ID 1. All worker processes have IDs higher than 1. The `myid()` function returns a process's ID.

Julia can also run in parallel on whole clusters of computers. A parallel cluster requires a password-free SSH environment. Julia then adds individual cluster computers using the `machinefile` option.

Anyone wanting to have a task executed by a particular process needs to use the `remotecall()` low-level function:

```
julia> result = remotecall(3, +, 2, 2)
RemoteRef(3,1,6)
```

The process with ID 3 calculates 2 + 2, a trivial example. However, the immediate response isn't the result, but rather a remote reference (`RemoteRef`), because the result isn't available locally as process 3 calculated it. To receive the result, you must first collect it from the remote process via `fetch()`:

```
julia> fetch(result)
4
```

As mentioned earlier, `remotecall()` is a low-level function. If it doesn't matter which process executes, you can use Julia's `spawn` macro to send tasks to other processes. The `spawnat` macro also makes it possible to specify the process. For example,

```
julia> @spawnat 2 rand(10,10)
```

creates a random matrix through process 2, and

```
julia> @spawn rand(10,10)
```

leaves the choice of process to Julia. Depending on the algorithm and problem, programmers can therefore decide whether to perform the process distribution themselves or to leave it totally to Julia.

It is usually easiest to let Julia get on with it. `spawn` also only returns a remote reference, which then again requires a `fetch()` call to collect the result:

```
julia> result = @spawn 2+2
RemoteRef(2,1,6)
julia> fetch(result)
4
```

Calling `remotecall()` or `spawn` doesn't guarantee that the immediately-returned remote reference will also contain the result. Only `fetch()` ensures this. For example, `spawn` could start a complex and lengthy computation using another process. Nevertheless, `spawn` won't initially block anything.

However, `fetch()` can only deliver the result once the calculation is also finished. That means it is `fetch()` that then blocks for a time and the local process waits until the result is available.

Distributed Objects

Julia also offers the possibility to directly generate distributed objects. This feature is particularly helpful when working with large matrices. Julia makes available distributed arrays that are of type `Darray`. To use these arrays, users first need to install the necessary package through the internal package management (Pkg):

```
julia> Pkg.checkout("DistributedArrays")
```

This command downloads and installs the `DistributedArrays` code. You can get an overview of all installed packages using `Pkg.status()`. If updates for packages are available, you can easily install them using `Pkg.Update()`. To use distributed arrays, you need to ensure that all processes load the package:

```
julia> @everywhere using DistributedArrays
```

The `everywhere` macro loads the package via `using` and ensures that it is available to all processes.

Julia offers numerous ways to generate distributed arrays. The easiest way is using the following functions:

```
dzeros(100,100,10)
dones(100,100,10)
drand(100,100,10)
drandn(100,100,10)
dfill(x,100,100,10)
```

These functions provide distributed arrays with the specified dimensions and characteristics. For example, `dzeros` provides an array filled with zeros. Julia sorts all this work out: The individual matrix elements are then distributed to the various processes and initialized.

Julia provides other useful functions for working with distributed arrays. The `localpart()` function, for example, returns part of the array that is assigned to the local process. `localindexes()` analogously returns the indices of the local part of the array.

If the local process is supposed to edit a distributed array, programmers can use `convert()`. `distribute()` converts a local array into a distributed array (i.e., the function distributes the array's data to the existing processes).

1 2 3 Next »

Buy this article as PDF

Express-Checkout as PDF

Price $2.95

(incl. VAT)

Check out with PayPal
The safer, easier way to pay

Buy Linux Magazine

SINGLE ISSUES

Print Issues

Digital Issues

SUBSCRIPTIONS

Print Subs

Digisubs

TABLET & SMARTPHONE APPS

Download on the App Store

GET IT ON Google play
US / Canada

GET IT ON Google play
UK / Australia

Related content

- Linux News
  - **Android gains market share**
  - **VMware Workstation**
  - **Perforce on Demand**
  - **openSUSE 12.2 released**
  - **CUDA 5 rc**
  - **Web Foundation index measures world**
  - **New Z shell release first since 2004**

more »

- Flam3 and Fractal Fr0st

Few fractal algorithms create as beautiful and ethereal structures as Flam3. The Fr0st GUI helps you master the complex software.

more »

- Linux Mint 10 "Julia" is now available!

The popular, Ubuntu 10.10 based, Linux distribution released Linux Mint 10 today.

more »

- OpenMP

OpenMP brings the power of multiprocessing to your C, C++, and Fortran programs.

more »

- New C++ Features in GCC

Recent versions of the GNU compiler include new features from the next C++ standard.

more »

1 Comment Linux Magazine

1 Login

Recommend Share

Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

best essay writing service • 2 years ago

Using this kind of programming language looks pretty effective and useful that there might be a lot of people out there who will be willing to try this thing and be able to have a better output than before.

• Reply • Share ›

Subscribe Add Disqus to your siteAdd DisqusAdd Privacy

Visit Our Shop

TRIAL SUBSCRIPTION

DIGITAL SUBSCRIPTION

SUBSCRIBE

Direct Download

Read full article as PDF:

Check out with PayPal The safer, easier way to pay Price $2.95

DevOps

- **Reinvent your network with DevOps tools and techniques:**

  Common DevOps Mistakes

  Packaging Apps To Run on Any Linux Device

  ZAP Provides Automated Security Tests in Continuous Integration Pipelines

  DevOps with DebOps

  Opportunities and Risks: Containers for DevOps

  Are you ready for DevOps? Try your luck with the beta version of Linux Professional Institute's new DevOps exam.

GPU Computing

- **News and views on the GPU revolution in HPC and Big Data:**

  HIP: CUDA Integration with ROCm

  Freeing the GPU

  Exploring AMD's Ambitious ROCm Initiative

  Crypto-Currency Mining Drives a Surge in GPU Sales

  New Platform for GPU-Enabled HPC and Ultra-Scale Computing

  Radeon Instinct: Evolving, Adapting & Learning Evolving, Adapting & Learning

News

Linus Torvalds' Precious Advice to Security Experts