Julia: A Fresh approach to parallel computing

Dr. Viral B. Shah

Intel HPC Devcon, Salt Lake City
Nov 2016
Opportunity: Modernize data science

The last 25 years...

Today’s computing landscape:

- Develop new learning algorithms
- Run them in parallel on large datasets
- Leverage accelerators like GPUs, Xeon Phis
- Embed into intelligent products

“Business as usual” will simply not do!
Those who convert ideas to products fastest will win

**Typical Workflow in the last 25 years**

1. **Develop Algorithms** (Python, R, SAS, Matlab)
2. **Rewrite in a systems language** (C++, C#, Java)
3. **Deploy**

**Compress the innovation cycle with Julia**

1. **Develop Algorithms** (Julia)
2. **Deploy**
JuliaCon 2016: 50 talks and 250 attendees
The user base is doubling every 9 months

**GitHub**
- 500 contributors
- 8,000 stars

**Julia Community**
- 150,000 users

**Website**
- 2,000,000 visitors

**YouTube**
- 200,000 views

**JuliaCon**
- 800 attendees

**Julia Packages**
- 1,200 packages
- 20,000 GitHub stars
The Julia community: 150,000 users
The Julia IDE
The Julia community is doing amazing things.
We want you in on it!

Sign in via LinkedIn
Sign in via GitHub
Sign in via Google

Jupyter
Create Jupyter Notebooks and share them.

Console
Use in-browser terminal emulator to fully control your Docker instance.

Google Drive
Collaborate with others. Sync notebooks and data via Google Drive.

Sync & Share
Setup folders to sync with remote git repositories.
Machine Learning: Write once Run everywhere

Many machine learning frameworks

- TensorFlow
- dmlc
- mxnet
- scikit-learn
- Merlin.jl
- Mocha.jl
- Knet.jl

Run on hardware of your choice

- Intel Core i7
- ARM
- NVIDIA CUDA
- OpenPOWER
- OpenCL
A few of the organizations using Julia
Traction across industries

Finance: Economic Models at the NY Fed

Liberty Street Economics

Engineering: Air Collision Avoidance for FAA

Resolution Advisories

Retail: Modeling Healthcare Demand

Robotics: Self-driving Cars at UC Berkeley
# Sequential method for calculating pi

```
function findpi(n)
    inside = 0
    for i = 1:n
        x = rand(); y = rand()
        inside += (x^2 + y^2) <= 1
    end
    4 * inside / n
end
```

```
julia> findpi(10000);
elapsed time: 0.0001 seconds
```

```
julia> findpi(100_000_000);
elapsed time: 1.02 seconds
```

```
julia> findpi(1_000_000_000);
elapsed time: 10.2 seconds
```

# Parallel method for calculating pi

```
addprocs(40);
```

```
@everywhere function findpi(n)
    inside = 0
    for i = 1:n
        x = rand(); y = rand()
        inside += (x^2 + y^2) <= 1
    end
    4 * inside / n
end
```

```
pfindpi(N)= mean( pmap(n->findpi(n), [N/nworkers() for i=1:nworkers()]) );
```

```
julia> pfindpi(1_000_000_000);
elapsed time: 0.33 seconds
```

```
julia> pfindpi(10_000_000_000);
elapsed time: 3.21 seconds
```

```
julia> pfindpi(100_000_000_000);
elapsed time: 10.2 seconds
```

```
julia> pfindpi(1_000_000_000_000);
elapsed time: 10.2 seconds
```

```
julia> pfindpi(10_000_000_000_000);
elapsed time: 3.21 seconds
```

```
julia> pfindpi(100_000_000_000_000);
elapsed time: 10.2 seconds
```

```
julia> pfindpi(1_000_000_000_000_000);
elapsed time: 10.2 seconds
```

```
julia> pfindpi(10_000_000_000_000_000);
elapsed time: 3.21 seconds
```

```
julia> pfindpi(100_000_000_000_000_000);
elapsed time: 10.2 seconds
```

```
julia> pfindpi(1_000_000_000_000_000_000);
elapsed time: 10.2 seconds
```

```
julia> pfindpi(10_000_000_000_000_000_000);
elapsed time: 3.21 seconds
```

```
julia> pfindpi(100_000_000_000_000_000_000);
elapsed time: 10.2 seconds
```
Not restricted to Monte Carlo only

A hierarchy of parallel constructs:

- Multiprocessing to go across a cluster
- Multithreading on the same node
- Concurrency within a process for I/O bound code
- Instruction level parallelization with SIMD codegen

Composable parallel programming model comprising of:

- Single thread of control
- SPMD with messaging (designed for multiple transports)
Case Study 1
Parallel Recommendation Engines

- **RecSys.jl** - Large movie data set (500 million parameters)
- Distributed Alternating Least Squares SVD-based model executed in Julia and in Spark
  - Faster:
    - Original code in Scala
    - Distributed Julia nearly 2x faster than Spark
  - Better:
    - Julia code is significantly more readable
    - Easy to maintain and update

Case Study 2
Enhancing Sky Surveys with Celeste.jl

Learning an Astronomical Catalog of the Visible Universe through Scalable Bayesian Inference

Jeffrey Regier*, Kiran Pannany†, Ryan Giordano‡, Rollin Thomas§, David Schlegel¶, Jon McAuliffe‖ and Prabhat¶
*Department of Electrical Engineering and Computer Science, University of California, Berkeley
†Parallel Computing Lab, Intel Corporation
‡Department of Statistics, University of California, Berkeley
§Physics Division, Lawrence Berkeley National Laboratory
¶NERSC, Lawrence Berkeley National Laboratory
Celeste: A Generative Model of the Universe

Aim: produce a comprehensive catalog of everything in the sky, using all available data.

- Infer properties of stars, galaxies, quasars from the combination of all available telescope data
- Largest Graphical Model in Science: $O(10^9)$ pixels of telescope image data; $O(10^6)$ vertices; $O(10^8)$ edges.
A Generative Model for Astronomical Images

- Each pixel intensity $x_{nm}$ is modeled as an observed Poisson random variable, whose rate is a deterministic function of latent random variables describing the celestial bodies nearby.

- Inference methods: variational Bayes (UC Berkeley), MCMC (Harvard)

- Celestial bodies’ locations determined 10% more accurately; Celestial bodies’ colors determined 50% more accurately (Regier, J., et al, ICML, 2015)
Celeste Scaling Results

A. *Single node, multi-threaded performance*

Figure 3: Multi-threaded performance. Strong-scaling Celeste on 154 light sources over up to 16 threads on a Cori Phase I node. Observe that scalability drops off beyond 4 threads; this is due to serial garbage collection.

Figure 4: Weak-scaling Celeste. While the total runtime shown is precise, the components of runtime shown are averaged across nodes and thus should be treated as being representative rather than exact.

Figure 5: Strong-scaling Celeste. Note the reduction in GC time correlates with reduction in runtime, whereas the increase in global arrays fetch time correlates with the increase in nodes.
Case Study 3
Stochastic Optimization of Energy Networks Economics

StructJuMP.jl - Cosmin G. Petra, Joey Huchette, Miles Lubin

- (150 LOC) Extension to JuMP.jl for modeling large-scale block-structured optimization problems across an HPC cluster
- Interface to PIPS-NLP optimization solver (300 LOC) for scale-out computation or Ipopt.jl for local computation
- Adds Benders Decomposition capabilities to JuMP

Results:

- StructJuMP.jl significantly decreases the time necessary for scalable model generation compared to competing modeling language approaches
- Successfully used for driving scale out optimization problem across a supercomputer at Argonne National Labs on over 350 processors

http://dl.acm.org/citation.cfm?id=2688224
Case Study 4

JINV - PDE Parameter Estimation

Lars Ruthotto, Eran Treister, Eldad Haber

- jInv.jl – A Flexible Framework for Parallel PDE Parameter Estimation
  - Provides functionality for solving inverse and ill-posed multiphysics problems found in geophysical simulation, medical imaging and nondestructive testing.
  - Incorporates both shared and distributed memory parallelism for forward linear PDE solvers and optimization routines

- Results:
  - Weak scaling tests on AWS EC2 (c4.large) instances achieves almost perfect scaling (minimum 93%)
  - Strong scaling tests on DC resistivity problem scales from 1 to 16 workers providing 5.5x overall problem speedup

A compiler framework on top of the Julia compiler for high-performance technical computing

- Identifies parallel patterns – array operations, stencils
- Applies parallelization, vectorization, fusion
- Generates OpenMP or LLVM IR
- Delivers 20-400x speedup over standard Julia
- #13 most popular Julia package out of >1000

Available at: https://github.com/IntelLabs/ParallelAccelerator.jl
Example: Black-Scholes

```julia
using ParallelAccelerator

@acc function blackscholes(sptprice::Array{Float64,1},
    strike::Array{Float64,1},
    rate::Array{Float64,1},
    volatility::Array{Float64,1},
    time::Array{Float64,1})

    logterm = log10(sptprice ./ strike)
    powterm = .5 .* volatility .* volatility
    den = volatility .* sqrt(time)
    d1 = (((rate .+ powterm) .* time) .+ logterm) ./ den
    d2 = d1 .- den
    NofXd1 = cndf2(d1)
    ...
    put = call .- futureValue .+ sptprice

end

put = blackscholes(sptprice, initStrike, rate, volatility, time)
```

Accelerate this function
Implicit parallelism exploited
Example (2): Gaussian blur

using ParallelAccelerator

@acc function blur(img::Array{Float32,2}, iterations::Int)
    buf = Array(Float32, size(img)...)  
    runStencil(buf, img, iterations, :oob_skip) do b, a
        b[0,0] =
        (a[-2,-2] * 0.003 + a[-1,-2] * 0.0133 + a[0,-2] * ... 
            a[-2,-1] * 0.0133 + a[-1,-1] * 0.0596 + a[0,-1] * ... 
            a[-2, 0] * 0.0219 + a[-1, 0] * 0.0983 + a[0, 0] * ... 
            a[-2, 1] * 0.0133 + a[-1, 1] * 0.0596 + a[0, 1] * ... 
            a[-2, 2] * 0.003 + a[-1, 2] * 0.0133 + a[0, 2] * ... 
        return a, b
    end
    return img
end

img = blur(img, iterations)
ParallelAccelerator.jl Performance

Evaluation Platform:
Intel(R) Xeon(R) E5-2699 v3
36 cores

ParallelAccelerator enables ~20-400× speedup over standard Julia
The future is automatic parallelism

HPAT.jl by Intel Parallel Computing Lab

High performance data analytics with scripting ease of use
- Translates data analytics Julia to optimized MPI
- Supports operations on arrays and data frames
- Automatically distributes data and generates communication
- Outperforms Python/MPI by >70x

Prototype available at:

https://github.com/IntelLabs/HPAT.jl
Matrix Multiply in HPAT

```julia
using HPAT
@acc hpat function p_mult(M_file, x_file, y_file)
    @partitioned(M, HPAT_2D)
    M = DataSource(Matrix{Float64}, HDF5, "/M", M_file)
    x = DataSource(Matrix{Float64}, HDF5, "/x", x_file)
    y = M*x
    y += 0.1*randn(size(y))
    DataSink(y, HDF5, "/y", y_file)
end
```

Generated code is 525 lines in MPI/C++
Set 96 indices for Parallel I/O

- Start, stride, count, block 2D indices for 4 hyperslabs, 3 matrices
Evaluation of Matrix Multiply

Also: 7x productivity improvement over MPI/Python
HPAT vs. Spark vs. MPI/C++

Speedup of HPAT over Spark in red

Cori at NERSC/LBL
64 nodes (2048 cores)
“If you have a choice of several languages, it is, all other things being equal, a mistake to program in anything but the most powerful one”.

- Paul Graham in *Beating the Averages*
  Co-Founder, Y-Combinator