Cataloging the Visible Universe through Bayesian Inference at Petascale in Julia

Keno Fischer
with the Celeste collaboration

August 28, 2019

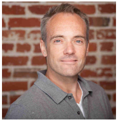
The Celeste.jl collaboration

Jeff Regier
UC Berkeley

Ryan Giordano
UC Berkeley

Steve Howard
UC Berkeley

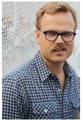
Max Lam
UC Berkeley



Jon McAuliffe
UC Berkeley

Keno Fischer
Julia Computing

Jarrett Revels
MIT



Andreas Noack
MIT/JC

Andy Miller
Harvard

Ryan Adams
Harvard

Kiran Pamnany
Intel

Debbie Bard
LBNL

Rollin Thomas
LBNL

David Schlegel
LBNL

Prabhat
LBNL

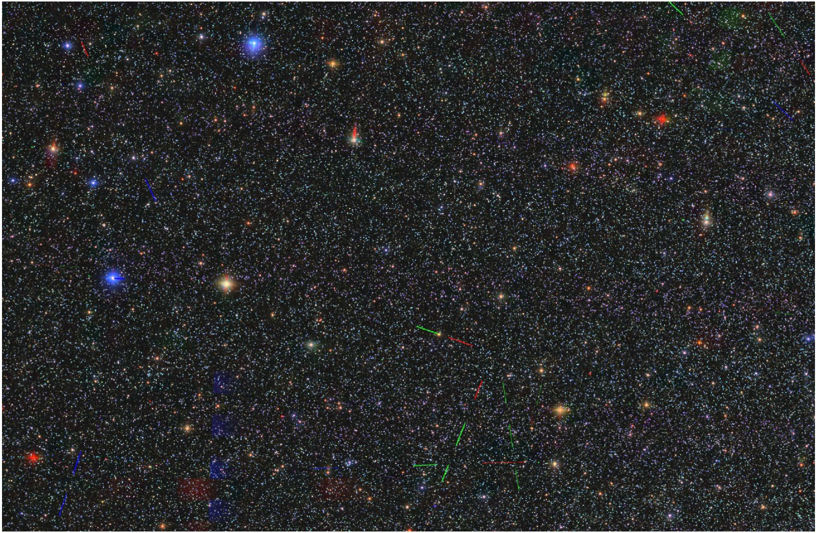
Celeste Accomplishments

1. First Julia application to exceed 1PF performance
   - Code ran on 9300 Cori Phase II nodes
   - 1.3 million threads on 650,000 KNL cores
2. Processed SDSS dataset in 15 minutes
   - Loaded and analyzed 178 TB
   - 188 million stars and galaxies
3. First comprehensive catalog of visible objects with state-of-the-art point and uncertainty estimates
4. Demonstration of Variational Inference on 8B parameters
   - 2 orders of magnitude larger than other reported results

The Data - An astronomical image

An image from the Sloan Digital Sky Survey, showing a galaxy from the constellation Serpens, 100 million light years from Earth, along with several other galaxies and many stars from our own galaxy.

One quarter square degree



An image from the Sloan Digital Sky Survey covering roughly one quarter square degree of the sky, containing roughly 10,000 detections.

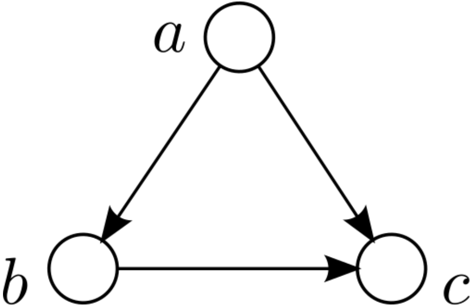
Faint light sources

Most light sources are near the detection limit.

Scientific goals

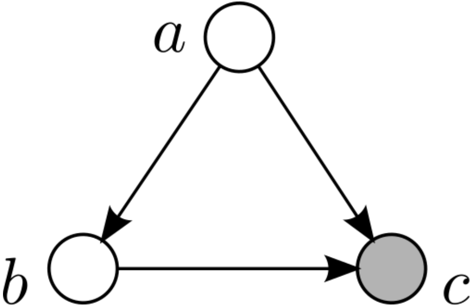
1. Catalog all galaxies and stars that are visible through the next generation of telescopes.
    - The Large Synoptic Survey Telescope will house a 3200-megapixel camera producing 15 terabytes of images nightly.
2. Replace non-statistical approaches to building astronomical catalogs from photometric data.
3. Identify promising galaxies for spectrograph targeting.
    - Better understand dark energy and the geometry of the universe.
4. Develop an extensible model and inference procedure, for use by the astronomical community.
    - Future applications might include finding supernovae and detecting near-Earth asteroids.

Our approach: graphical models



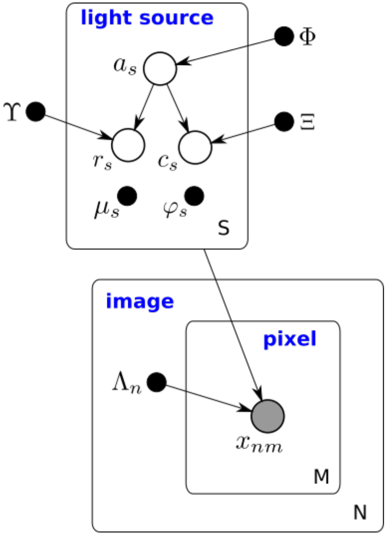
$$p(a, b, c) = p(c|a, b)p(b|a)p(a)$$

Our approach: posterior inference



$$\underbrace{p(a, b|c)}_{\text{posterior}} \propto \underbrace{p(c|a, b)}_{\text{likelihood}} \underbrace{p(b|a)p(a)}_{\text{prior}}$$

The Celeste.jl graphical model



Intractable posterior

Let $\Theta = (a_s, r_s, c_s)_{s=1}^{S}$. The posterior on Θ is intractable because of coupling between the sources:

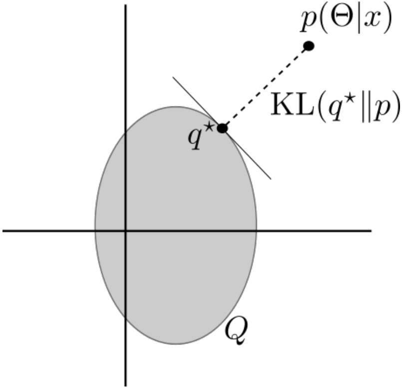
$$p(\Theta|x) = \frac{p(x|\Theta)p(\Theta)}{p(x)}$$

and

$$\begin{aligned} p(x) &= \int p(x|\Theta)p(\Theta)\, d\Theta \\ &= \int \prod_{n=1}^{N} \prod_{b=1}^{B} \prod_{m=1}^{M} p(x_{nbm}|\Theta)p(\Theta)\, d\Theta. \end{aligned}$$

Variational inference

Variational inference approximates the exact posterior p with a simpler distribution $q^\star \in Q$.



Average error for light sources from Stripe 82

		Photo	Celeste.jl
Position	(pixels)	0.36	**0.27**
Brightness	(magnitude)	0.21	**0.14**
Color u-g	(magnitude)	1.32	**0.60**
Color g-r	(magnitude)	0.48	**0.21**
Color r-i	(magnitude)	0.25	**0.12**
Color i-z	(magnitude)	0.48	**0.17**
Profile	(proportion)	0.38	**0.28**
Axis ratio	(magnitude)	0.31	**0.23**
Scale	(arcseconds)	1.62	0.92
Angle	(degrees)	22.54	**17.54**

Lower is better. Highlighted results are better by more than 2 standard deviations.

Julia makes implementing complex objective functions possible.

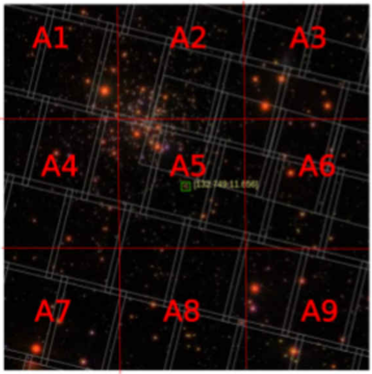
$$
\begin{aligned}
&\mathcal{L}\left(\chi, \mu, \kappa, \gamma, \zeta, \beta, \lambda, \theta, \rho, \sigma, \varphi\right) \\
&= C + \sum_{n=1}^{N} \sum_{b=1}^{B} \sum_{m=1}^{M} \Bigg\{ \sum_{a \in \{0,1\}^S} \prod_{s=1}^{S} \chi_s^{a_s} \left(1 - \chi_s\right)^{1-a_s} \Bigg\{ \int_{r_1} \int_{c_1} \int_{k_1} \cdots \int_{r_S} \int_{c_S} \int_{k_S} \\
&\qquad x_{nbm} \log \Bigg[\epsilon_{nb} + \sum_{s=1}^{S} r_s \prod_{j=b}^{b_r} \exp\left\{c_{sb}\right\} \prod_{j=b_r}^{b-1} \exp\left\{c_{sb}\right\} \\
&\qquad\qquad \times \int \sum_{k=1}^{3} \bar{\alpha}_{nk} \phi\left(m - w; \bar{\xi}_{nbk}, \bar{\Sigma}_{nbk}\right) g_{si}\left(w\right) dw \Bigg] \\
&\qquad - \iota_{nb} \sum_{s=1}^{S} r_s \prod_{j=b}^{b_r} \exp\left\{c_{sb}\right\} \prod_{j=b_r}^{b-1} \exp\left\{c_{sb}\right\} \int \sum_{k=1}^{3} \bar{\alpha}_{nk} \phi\left(m - w; \bar{\xi}_{nbk}, \bar{\Sigma}_{nbk}\right) g_{si}\left(w\right) dw \\
&\quad dr_1 \; dc_1 \; dk_1 \; \ldots \; dr_S \; dc_S \; dk_S \Bigg\} \Bigg\} \\
&\quad - \sum_{s=1}^{S} \Bigg\{ D_{\mathrm{KL}}\left(q(a_s), p(a_s)\right) + \sum_{i=1}^{2} \chi_s^{a_s} \left(1 - \chi_s\right)^{1-a_s} \\
&\qquad \times \left[D_{\mathrm{KL}}\left(q(r_s|a_s = i), p(r_s|a_s = i)\right) + D_{\mathrm{KL}}\left(q\left(k_s, c_s|a_s = i\right), p_s\left(k_s, c_s|a_s = i\right)\right)\right] \Bigg\}.
\end{aligned}
$$

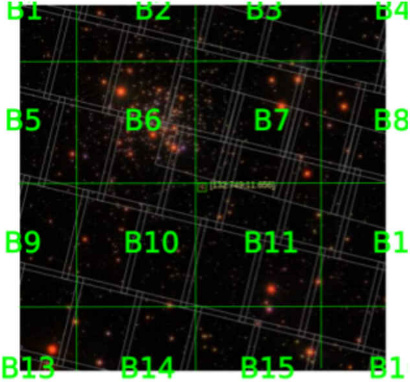
At scale, variational inference is distributed optimization.

We find a **stationary point** through a "hardware-aware" multi-level numerical optimization scheme.

1. Compute nodes simultaneously optimize disjoint regions of the sky (block coordinate ascent).
2. Each node's threads simultaneously optimize non-overlapping light sources (block coordinate ascent).
3. Each light sources is optimized by Newton's method with a trust region constraint.

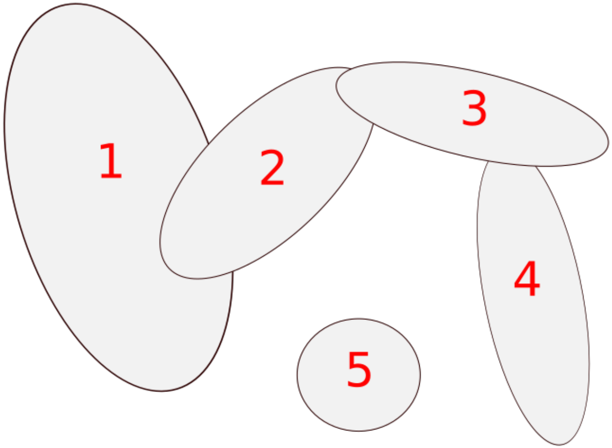
Parallelism among nodes





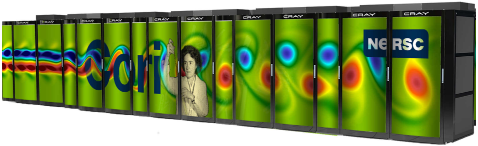
A region of the sky, shown twice, divided into 25 overlapping boxes: A1,. . .,A9 and B1,. . .,B16. Each box corresponds to a task: to optimize all the light sources within its boundaries.

Parallelism among threads



Light sources that do not overlap may be updated concurrently.

Target platform: Cori Phase 2



9,300 nodes

× 68 KNL cores / node

× 8-wide DP SIMD Lane / core

× 2 FLOPs / FMA instruction

× 2 FMA instructions / cycle

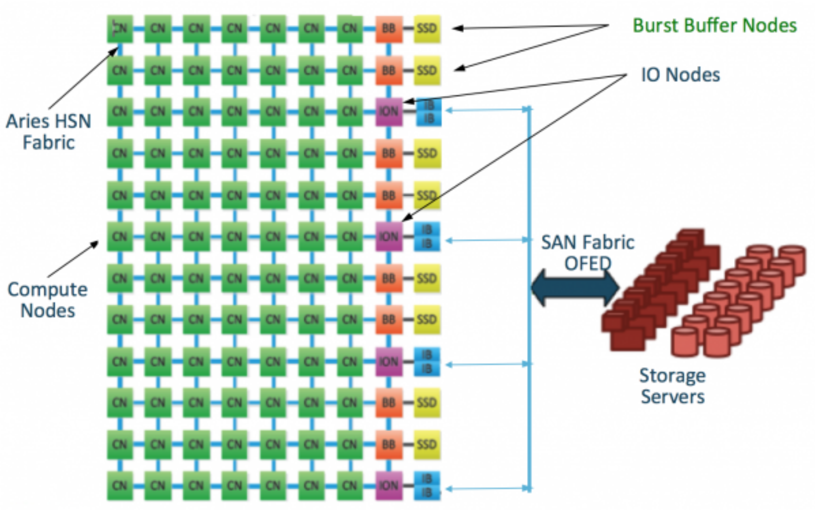
× 1.4 GHz (cycles / second)

= 30 PetaFlops

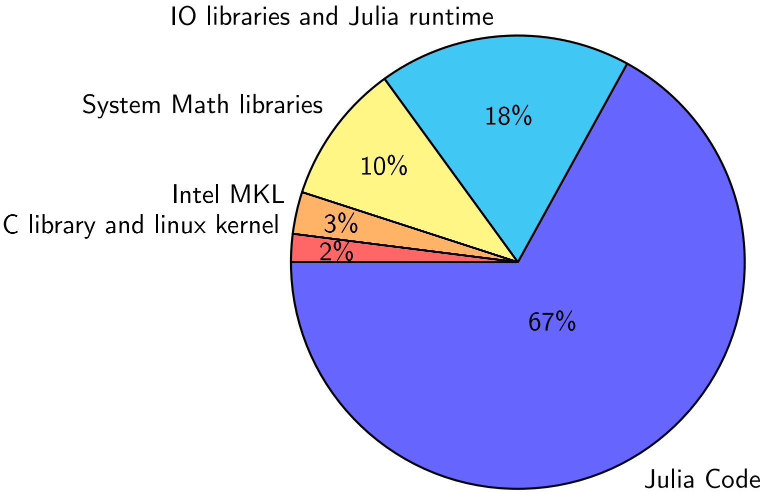
Scaling results: Ready for LSST

- 1.54 PetaFlops peak performance in native Julia, using 1.3 million threads on 650,000 Intel Xeon Phi cores
- 14.6 minutes to process most of SDSS, loading 178 TB and optimizing 188M stars and galaxies
- "sixteenth degree" benchmark: from 320 seconds (Early March, 2017) to 17 seconds (Mid April, 2017)

Solving large scale I/O problems



Proportional Time spent



Julia in a slide

- High level, dynamic (but with strong notion of types)
- Multiple dispatch based
- Modern Features: Lisp-like macros, Garbage Collection
- Built-in support for parallelism (distributed memory, shared memory, vectorization)
- Wide architecture support (Built on LLVM => High performance support for many CPU architectures as well as several GPU architectures)
- Fast :)

Writing High Performance julia code

1. Follow the performance guidlines (no global state/eval/etc in hot parts)
2. Type stability
3. Cutting down on dynamic memory allocations
4. Use the profiler to find where you're spending your time
5. Use the memory profiler to find allocation (dual benefit - less time spent allocating/less time in GC)

The life of a multiply

```
for #= optimization steps =#
# function elbo_likelihood(...)
for n in 1:ea.N # images
for s in ea.active_sources # active sources
for w2 in 1:W2, h2 in 1:H2 # pixels
# function add_pixel_term!
for s in 1:ea.S # active and inactive sources
# [2 more functions]
for i = 1:2 # Galaxy types
for j in 1:8 # Galaxy component
for k = 1:size(gal_mcs, 1)
# [3 more function calls]
for shape_id in 1:length(gal_shape_ids)
for u_id in 1:2
for sig_id in 1:3
for x_id in 1:2
  bvn_us_h[u_id, shape_id] +=
    bvn_xsig_h[x_id, sig_id] *
    sig_sf_j[sig_id, shape_id] *
 (-wcs_jacobian[x_id, u_id])
```

ILP $\sim$ 800

ILP $\sim$ 10 – 20

How much ILP is required?

HSW:

- Throughput: 2/cycle
- Latency: 5 Cycles
- Vector Width: 4

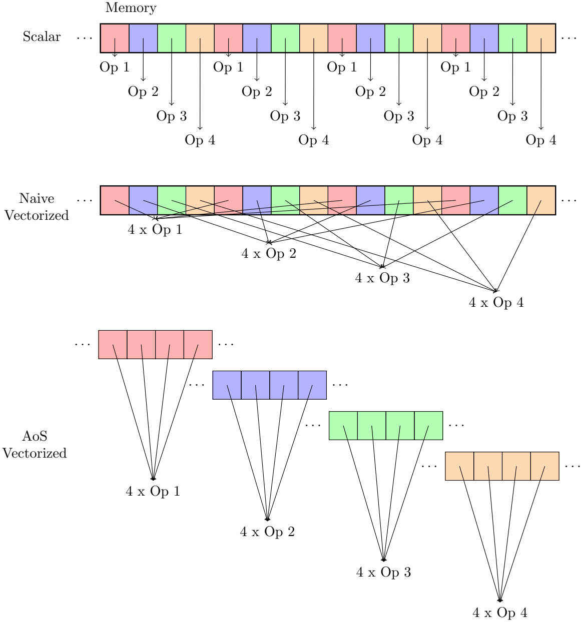
Need ILP of 40 for max throughput
Register space becomes limitation

KNL:

- Throughput: 2/cycle
- Latency: 6 Cycles
- Vector Width: 8

Need ILP of 96 for max throughput

Going beyond - SoA transformation



StructOfArrays.jl package changes storage order, but retains abstraction.
Change one line, compiler does the rest.
Before:

```
struct BvnBundle{T<:Real, HasGradient, HasHessian}
    bvn_derivs::BivariateNormalDerivatives{T}
    star_mcs::Matrix{BvnComponent{T}}
    gal_mcs::Matrix{GalaxyCacheComponent{T}}
    sbs::Vector{SourceBrightness{T,
                HasGradient, HasHessian}}
end
```

After:

```
struct BvnBundle{T<:Real, HasGradient, HasHessian}
    bvn_derivs::BivariateNormalDerivatives{T}
    star_mcs::Matrix{BvnComponent{T}}
    gal_mcs::similar(StructOfArrays,
                     Matrix{GalaxyCacheComponent{T}})
    sbs::Vector{SourceBrightness{T,
         HasGradient, HasHessian}}
end
```

Optimization Takeaways

1. Encoding semantics separately from representation makes optimizing memory layout a one-line change
2. Julia makes this easy (very powerful meta programming)
3. Whole code for highly-generic framework for representing abstract indices/generating code for sparse structure ~ 200 lines of julia code (could be split out into a separate package)
4. Large performance gains, without any changes to the algorithm
5. Easily switch back to known-good data representations for debugging

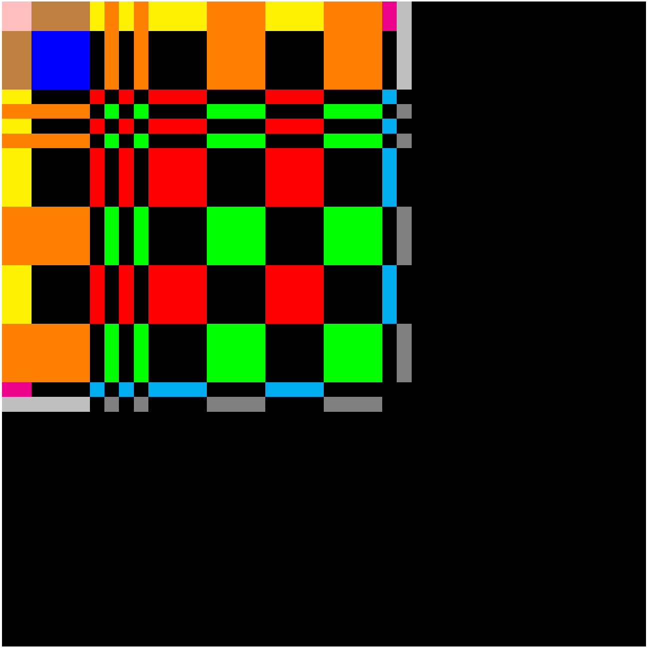
Conclusions

1. First Julia application to exceed 1PF performance
   - Single-node optimizations
   - Multi-node scaling considerations
2. Processed SDSS dataset in 15 minutes
   - Overcame significant I/O challenges to process 178 TB
3. First comprehensive catalog of visible objects with state-of-the-art point and uncertainty estimates
   - 188M stars and galaxies
4. Quality of generated native code suitable for HPC requirements and getting better

Questions?

Backup slides

Going beyond - Exploiting Locality/Sparsity



```
E_G_s.h[ids.u, ids.u] += (a_i * sb_E_l_a_b_i_v) *
        fsm_i.h[ids.u, ids.u]
```

`ids.u` is an array of indices into the matrix, indicating where the relevant data is stored:

```
julia> Celeste.Model.ids.u
[1, 2]
```

Idea: Make `ids.u` an object and use multiple dispatch to encode the storage location for data.

```
julia> Celeste.Model.ids.u
Param{:u, (2,)}()

# Automatically generate these
getindex(M::Matrix, ::Param{:u}) = M[[1,2]]
getindex(M::SparseStruct, ::Param{:u}) = M.u_data
```

SparseStruct memory layout