Speeding up computation using Julia: an illustration using discrete choice demand models.

Joris Pinkse, with Andy Tang

Plan

- why **I** use Julia;
  - *I'm not proselytizing*
- my projects in Econometrics / Industrial Organization

Context

- new econometric techniques for structural models
- collaborations with coauthors in Industrial Organization who have large data sets (e.g. 20 million individuals)
- want others to use my econometric methodology
- have to make realistic hardware expectations
- budgetary constraints
- I am **not** a computer scientist

Speed determinants

- hardware
  - *GPUs / CPUs, memory,...*
- libraries / packages
  - *Flux, PyTorch, BLAS, KNITRO*
- language
  - *C, Rust, Julia, Matlab, Python, R*
- programmer ability
  - *differences in programmer ability can exceed intrinsic differences across languages*
- time commitment

Approaches

compilation: C, Rust

interpretation: Matlab, Python, Perl

just in time compilation: Julia

- *Languages can use a mixture of approaches*

What I (dis)like about Julia:

😄 fast

😄 relatively easy to figure out what slows you down

😄 intuitive

😄 multiple dispatch

😄 open source

😄 packaging system

😄 free

😄 can create Julia code while running Julia code

☹️ ensure type stability

☹️ JIT is a double-edged sword

☹️ constraints imposed by others

Julia is fast

- every language is converted to machine instructions
- speed of a language reflects how good this conversion is
- some languages are more suitable than others
- compilation is faster than interpretation
- Julia uses Just-In-Time (JIT) compilation

Toy speed comparisons

- less meaningful than larger sets of operations
  - *my Python, R, Rust is not that good*
- differences between fast and slow languages are typically less pronounced

Standard time-consuming operations

- don't expect large differences for large matrices
  - *for large matrices, speed is determined by BLAS choice*
  - *one can set the BLAS choice in most languages*
  - *all timing comparisons are net of compilation*

matrix inversion — single core

	Julia	Numpy	Numba +Scipy	R	Rust +openblas
100×100 (μs)	154	405	421	1,000	164
1000 x 1000 (ms)	48	38	50	92	51

Sums

- this is where compiled languages shine

2-regressor OLS estimation using sums — single core

	C	Julia	Numpy	Numba +Scipy	R	Rust
μs	0.08	0.08	0.95	0.13	0.92	0.08

How you do things is important

diag(X * Y) .* Z (parallel, μs)

	C	Julia	Numpy	Numba +Scipy	R	Rust +faer
matrix op	N/A	126	145	117	244	80
for loops	14	68	10,826	86	3,297	87
einsum	20	13	41	N/A	176	80

- *differences on this slide were less pronounced on Andy's computer than mine*

Speed conclusion

- a skilled programmer in a slower language can often create code that runs faster than a less skilled programmer using a faster language
- but, conditional on skill, a programmer has to do less in Julia than in other languages to achieve good performance
- there are fewer performance cliffs
- the more complex the task, the harder it is to make 'slower' languages perform well

Some generic suggestions

- make a tradeoff between your time and computing time
- use GPUs if you have good ones
- pay attention to language-specific hints
- think about what will be slow and what will be fast
- don't recompute the same thing
- use optimized packages
- use parallelization
- reuse memory
- use language-specific profiling tools
- processor-bound versus memory bound
- make your code cache-friendly

Syntax and such

Julia	**Numpy**	**meaning**
X’	X.T	X'
X^ -1	numpy.linalg.inv(X)	X^{-1}
X^(1/3)	scipy.linalg.fractional_matrix_power(X, 1/3)	$X^{1/3}$
X = randn(3, 5)	X = numpy.random.normal(size = (3,5))	
A .= diag(X * Y) .* Z	A[:] = numpy.diag(X @ Y)[:, numpy.newaxis] * Z	
@tullio V[i] := X[i,j] * Y[j,i]	v = numpy.einsum(’ij,ji->i’, X, Y)	$v_i = \sum_j x_{ij} y_{ji}$

Multiple dispatch

- multiple methods, same name, different arguments

```
function f( x, y )
    ...
end

function f( x :: Float64, y :: Int )
    ...
end
```

- Julia figures out which one to call at runtime
  - *so no need to define separate functions f_int, f_float, etc*
- in a generic definition of f, Julia generates specialized methods for each type

```
function f( x, y )
    x^5 * y
end
```

- calls to f(1,2), f(1.0,2.0) and f(“hello ”, “there”) generate three different methods, one for integers, one for floats, and one for strings, each optimized for their type.

Multiple dispatch — why I care

- (unpublished) package for methods used in industrial organization
- includes code to compute estimates for different demand models
- some code overlaps (gets reused), some doesn't
- user can override without changing the package
- users may require different degrees of precision
- all I need to do is to define methods with desired level of generality
  - *e.g. some for all discrete choice demand models, some for random coefficients discrete choice demand models, etc*

Multiple dispatch cont'd

- four basic commands:

  Estimate! computes estimates

  Infer! computes test statistics and such

  Summarize! computes e.g. diversions and elasticities

  Experiment! does e.g. welfare calculations or a merger simulation
- structure of user calls is the same

```
    results = Estimate!( D )
    Infer!( results )
    Summarize!( results )
```

Multiple dispatch cont'd

- what changes is the argument D

```
D = Dict(
    :model                => :rclindem,
    :estimator            => :mlegmm,
    :productdata          => "loadfakedata/50/products.csv",
    :consumerdata         => "loadfakedata/50/consumers.csv",
    :marketsizedata       => "loadfakedata/50/marketsizes.csv",
    :demographicdraws     => "loadfakedata/50/draws.csv".
    :choices              => :choice,
    :shares               => :share,
    :markets              => :market,
    :marketsizes          => :N,
    :products             => :product,
    :outsidegood          => "outside",
    :demographics         => [ :income, :income, :age ],
    :xzcharacteristics    => [ :constant, :ibu, :ibu ],
    :xvcharacteristics    => [ :ibu, :abv ],
    :regressors           => [ :constant, :ibu, :abv ],
    :instruments          => [ :constant, :ibu, :abv ]
    )
```

- all that you need to do to estimate a complete random coefficients discrete choice demand model with both micro and macro data
- for other estimators / models, the structure is the same

Example timing: my 6.5 year old computer at home

- optimization only

MLE, random coefficients demand model with 50k consumers, 50 markets, 950 products (≈microblp)	6s
MLE, random coefficients demand model with 5k consumers, 500 markets, 100k products (≈microblp)	8m
MLE, latent consumer groups demand model with 250k consumers, 100 markets, 2k products, ngroups = 3	1s

What will be in the package?

demand estimation: (mostly done)

- **CLEER** (Grieco, Murry, Pinkse, Sagl)
- **latent groups estimator** (Pinkse, Ren, Setzler)
- **BLP of various kinds** (Berry, Levinsohn, Pakes)
- **multinomial logit**
- **mixed logit**
- **nested logit**
- **nonlinear versions of the above**
- **nonparametric** (Pinkse, Slade, Brett; Compiani; Brand, Smith)
- **etcetera**

production functions: (not started)

auctions: (not started)

dynamic games: (not started)

What if you don’t want/need/like Julia

- front ends for Python, R

Conclusions

- if it is just for you and you have $$$, consider a GPU-based system
- if there is an existing high performance library that does what you want $\longrightarrow$ great
  - *you wouldn't be at this conference*
- if there is a reliable package for your language that is fast enough $\longrightarrow$ use it
  - *nudge*
- consider using a performant language like Julia