Peformance and profiling in Julia

Fredrik Bagge Carlson[1]

[1]Dept. Automatic Control, Lund Institute of Technology
Lund University

Outline

1. Write code with performance in mind
2. Profile your code
3. Optimize your code

Step one, put your code in functions

A global variable might have its value, and therefore its type, change at any point. This makes it difficult for the compiler to optimize code using global variables.

$\Downarrow$

Any code that is performance critical or being benchmarked should be inside a function.

Avoid global variables (unless declared `const`)

```julia
PRINT = false
function foo()
  for i = 1:1_000_000_000
    if PRINT
      print(i)
    end
  end
end
@time foo()
  0.795711 seconds
```

```julia
const PRINT = false
function foo()
  for i = 1:1_000_000_000
    if PRINT
      print(i)
    end
  end
end
@time foo()
  0.000002 seconds
```

`PRINT` is `false` in both cases, but the compiler can rely on it in the second case

Type declarations, type stability

Useful as assertion for debugging, but does not make the code faster.

Exception: Declare specific types for fields of composite types

```
type Foo
    field
end
```

```
type Foo
    field::Type
end
```

It is in general bad for performance to change the type of a variable, type annotation will prevent this.

Type stability

An example of type instability

```
function foo()
    a = 1 # Int64
    for i = 1:100_000_000
        a += i/(i+1)
    end
    a
end
@time foo()
  2.167254 seconds
  (200.00 M allocations:
  2.980 GB,  22.27% gc time)
```

```
function bar()
    a = 1.0 # Float64
    for i = 1:100_000_000
        a += i/(i+1)
    end
    a
end
@time bar()
  0.715188 seconds
  (5 allocations: 176 bytes)
```

Julia uses column major convention

```julia
function foo()
   x = Array(Float64,
             (10_000,10_000))
   for i = 1:size(x,1)
      for j = 1:size(x,2)
         x[i,j] = i*j
      end
   end
end
@time foo()
  3.448774 seconds
```

```julia
function bar()
   x = Array(Float64,
             (10_000,10_000))
   for j = 1:size(x,2)
      for i = 1:size(x,1)
         x[i,j] = i*j
      end
   end
end
@time bar()
  0.300085 seconds
```

Think about this when you are choosing how to store your data!

Avoid unnecessary memory allocation

Julia passes arrays as references. Use this to re-use already allocated memory.

```
function food()
    A = Array(Int64,(100,100))
    for i = eachindex(A)
        A[i] = i
    end
    return A
end

function eat()
    for i = 1:10_000
        chicken = food()
        sum(chicken)
    end
end
@time eat()
  0.297590 seconds
(30.00 k allocations: 763.855 MB, 33.59% gc time)
```

```
function beer!(A)
    for i = eachindex(A)
        A[i] = i
    end
end

function drink()
    weiss = Array(Int64,(100,100))
    for i = 1:10_000
        beer!(weiss)
        sum(weiss)
    end
end

@time drink()
  0.060649 seconds
(7 allocations: 78.375 KB)
```

New plate every time, lots of time to clean! (garbage collect)

Use the same glass every time, drink more beer!

Profiling

Profiling

Your goto-tool is always `@time`, watch memory allocation and GC-time

- Type instability
- Allocations

Profiling

Julia has built in profiling capabilities

```
julia> @profile foo()

julia> Profile.print()
     23 client.jl; _start; line: 373
       23 client.jl; run_repl; line: 166
          23 client.jl; eval_user_input; line: 91
             23 profile.jl; anonymous; line: 14
                8  none; myfunc; line: 2
                 8 dSFMT.jl; dsfmt_gv_fill_array_close_open!; line: 12
                15 none; myfunc; line: 3
                 2  reduce.jl; max; line: 35
                 2  reduce.jl; max; line: 36
                 11 reduce.jl; max; line: 37
```

ProfileView

ProfileView package is nicer

```julia
using ProfileView
@profile foo()
ProfileView.view()
```

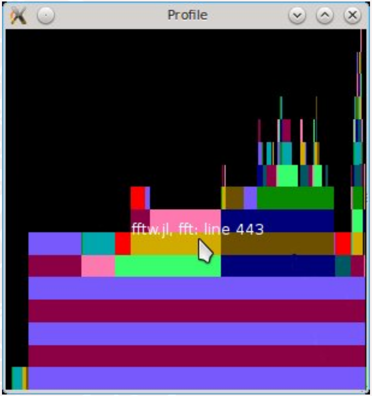

Figure: Image shamelessly borrowed from Tim Holy

https://github.com/timholy/ProfileView.jl

Profiling tools

```
julia --help
--code-coverage=none|user|all Count executions of
source lines (omitting setting is equivalent to
"user")
--track-allocation=none|user|all Count bytes
allocated by each source line
```

TypeCheck.jl

Optimization

Optimize your code

Use the result of `@profile, @time, track-allocation=user, code-coverage=user`

- If your code spends 50% doing garbage collection, you can reduce your running time with *up to* 50% by better memory management.

Devectorize

```
function foo(A)
    log(sum(exp(A)))
end
```

```
using Devectorize
function bar(A)
    @devec ret = log(sum(exp(A)))
    ret
end
```

```
function test(f::Function)
   A = ones(1_000_000)
    for i = 1:1000
        f(A)
    end
end

@time test(foo) 6.249262 seconds
@time test(bar) 2.714373 seconds
```

Performance enhancing start-up arguments

```
julia --help

--check-bounds=yes|no   Emit bounds checks always or
never (ignoring declarations)

--math-mode=ieee|fast   Disallow or enable unsafe
floating point optimizations (overrides @fastmath
declaration)

@inbounds

@fastmath
```

Julia startup arguments

```
julia -math-mode=fast
```

Type instability revisited

```julia
function foo()
    a = 1 # Int64
    for i = 1:100_000_000
        a += i/(i+1)
    end
    a
end
@time foo()
2.1672 sec # Without fastmath
1.8540 sec # With fastmath
```

```julia
function bar()
    a = 1.0 # Float64
    for i = 1:100_000_000
        a += i/(i+1)
    end
    a
end
@time bar()
0.7151 sec # Without fastmath
0.1911 sec # With fastmath
```

Warning: floating point operations are reordered and numerically unstable algorithms might fail

Benchmarking

- Put your code in functions
- Let the function compile before timing
- Do not put function definitions and test code in same file (unless precompilation is done before timing)
- Watch out for unexpected memory allocation.
- Read the performance tips!

Homework

Monte-Carlo simulation of a bootstrap particle filter

- I provide the baseline code
- My code provides a descent particle filter implementation
- The code is bad from a julia-performance point of view
- Your job is to optimize it
- Optimized code has to be equivalent (do not provide another implementation of a particle filter)

$$
\begin{aligned}
x^+ &= 0.5x + \frac{25x}{1+x^2} + 8\cos(1.2(t-1)) + w \\
y &= 0.05x^2 + v \\
w, v &\sim \mathcal{N}(0,\sigma_w),\ \mathcal{N}(0,\sigma_v) \quad E(wv^\mathsf{T}) = 0
\end{aligned}
$$

The particle filter

```
for t = 2:T # Main loop
    # Resample
    j = resample(w[t-1,:]')
    # Time update
    xp[t,:] = f(xpT,t-1) + sw*randn(1,N)
    # Measurement update
    w[t,:] = wT + g(y[t]-0.05xp[t,:].^2)
    # Normalize weights
    offset = maximum(w[t,:])
    normConstant = log(sum(exp(w[t,:]-offset)))+offset
    w[t,:] -= normConstant
end
```

⋮

The Monte-Carlo simulation

```julia
particle_count = [5 10 20 50 100 200 500 1000 10_000]
time_steps = [20, 50, 100, 200]
for (Ti,T) in enumerate(time_steps)
  for (Ni, N) in enumerate(particle_count)
    # Calculate how many Monte-Carlo runs to perform for the current
    # T,N configuration
    montecarlo_runs =
        maximum(particle_count)*maximum(time_steps) / T / N
    for mc_iter = 1:montecarlo_runs
      for t = 1:T-1 # Simulate one realization of the model
        x[t+1] = f(x[t],t) + sv*randn()
        y[t+1] = 0.05x[t+1]^2  + sv*randn()
      end # t
      xh = pf( y, N, g, f, sw0 ) # Run the particle filter
      RMS += rms(x-xh) # Store the error
    end # MC
```

⋮