Astro 585: HW 6

Codename: The Maxwell-Jüttner Distribution

March 9, 2014

My git repository is here: `https://github.com/hsgg/astro585`, clone URL `https://github.com/hsgg/astro585.git`.

1 Parallel stuff

1a) nprocs() is one larger than nworkers(). So there seems to be one process that manages all the ones that will do the actual computation.

1b) As you told us, the functions are only defined on the main process, not the workers.

1c) Presumably every worker writes to their own copy of 'integral', and the main thread doesn't do anything, so 'integral' in the main thread remains 0.0.

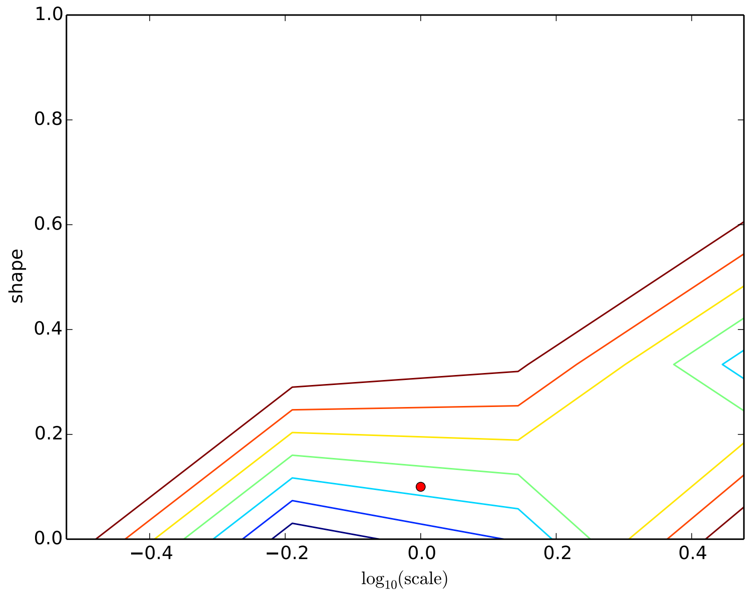
pmap(): Looks like the main process is using one thread (ca 95%), the other two share the other one, but they don't use as much (ca 40% each).

map(): Uses only one processor at a time. That's strange. The distributed array seems to make it really slow. Not distributing makes the operation finish in much less than a second, rather than 10s.

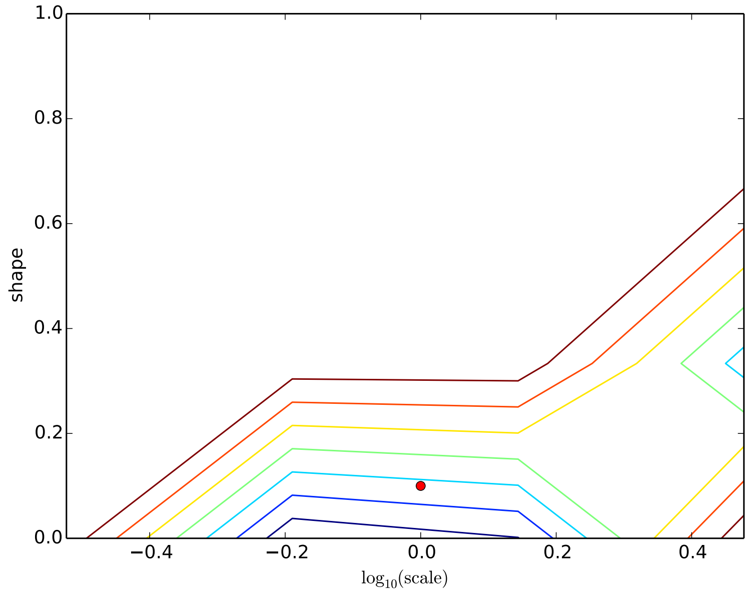
OK, I am not yet convinced of distributed arrays, although it seems to make sense. There is significant overhead involved with this.

2 Kepler

185 seconds when running in serial. Oddly, my plot looks differnt than yours when running your code:

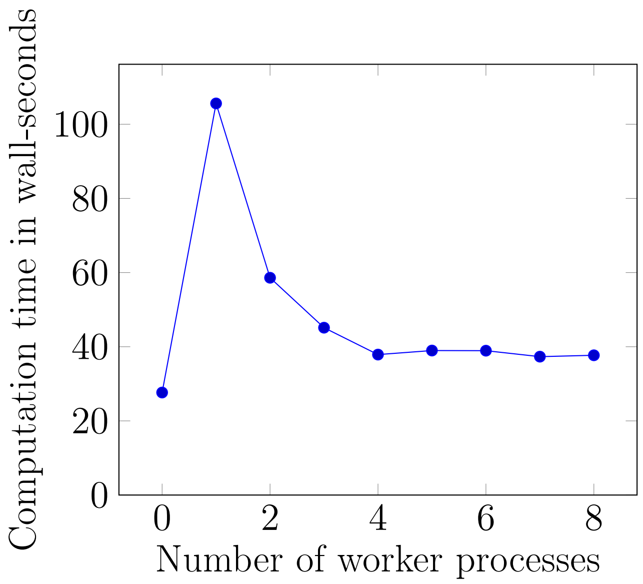


101 seconds in parallel with 2 cores, so slightly more than half the time, with almost the same result. There is a difference close to (0.0, 0.0). Not good:



Using distributed arrays wouldn't change the performance much, because there is little overhead due to communication. There are a total of 64 different points in parameter space to test, or 192 floats, and the return value is also just a list of 64 floats. Not much communication.

On workstation with 4 cores (yeah, I know, 4 != 8):



I am surprised by the fact that the single-threaded version is so much faster than even using 4 worker threads. I would not expect much communication overhead, since we are only transferring a few hundred floats from process to process.

Turning 'parameterspace' into a distributed array did not have any effect, either. I guess that is expected if there is not much data being communicated.

The code:

```
#!/usr/bin/env julia

#if nprocs() < 2
#  @time addprocs(2) # 13 seconds
#end
addprocs(4)

# Why does it take a minute just to load these? I hope they fix it in
# julia-0.3!
@time @everywhere using Distributions
@time @everywhere using PyPlot
@everywhere const days_in_year = 365.2425;
@time @everywhere include("HW6_Q2_planet_populations.jl")


@everywhere function eval_model_on_grid_parallel(etas::Array, shapes::Array,
    scales::Array, num_stars = 1600; num_evals = 1, true_eta = 0.2,
    true_shape = 0.1, true_scale = 1.0)
  const solar_radius_in_AU = 0.00464913034
  minP = (2.0*solar_radius_in_AU)^1.5
  maxP = 4*days_in_year/3
  data_obs = generate_transiting_planet_sample(true_eta, true_shape, true_scale, num_stars;
      minP=minP, maxP=4*days_in_year/3)
  stats_obs = compute_stats(data_obs)

  # This is ridiculous. Surely there is a better way to do this so that pmap()
  # or similar can create a grid from the three arrays 'etas', 'scales', and
  # 'shapes', and I don't need to do that on my own?
  parameterspace = Array((Float64, Float64, Float64),
      length(etas) * length(shapes) * length(scales))
  for k in 1:length(scales)
    for j in 1:length(shapes)
      for i in 1:length(etas)
        idx = (k-1) * length(etas) * length(shapes) + (j-1) * length(etas) + (i-1)
        parameterspace[idx + 1] = (etas[i], shapes[j], scales[k])
      end
    end
  end

  #parameterspace = distribute(parameterspace) # has no effect?

  result = pmap(pars -> evaluate_model(stats_obs, pars[1], pars[2], pars[3], num_stars;
            minP=minP, maxP=maxP, num_evals=num_evals), parameterspace)

  #result = Array(Float64, length(etas) * length(shapes) * length(scales))
  #for i in 1:length(result)
  #  x = parameterspace[i][1]
  #  y = parameterspace[i][2]
  #  z = parameterspace[i][3]
  #  println(i, ' ', x, ' ', y, ' ', z)
  #  result[i] = evaluate_model(stats_obs, x, y, z, num_stars;
  #      minP=minP, maxP=maxP, num_evals=num_evals)
  #end

  println(result)

  dist = Array(Float64, (length(etas), length(shapes), length(scales)))
  for k in 1:length(scales)
    for j in 1:length(shapes)
      for i in 1:length(etas)
        idx = (k-1) * length(etas) * length(shapes) + (j-1) * length(etas) + (i-1)
        dist[i, j, k] = result[idx + 1]
      end
    end
  end

  return dist
```

```
end



min_eta = 0.0
max_eta = 1.0
min_shape = 0.0001
max_shape = 1.0
min_log_scale = log10(0.3)
max_log_scale = log10(3.0)
num_eta = 4
num_shape = 4
num_scale = 4
num_evals = 1
etas = linspace(min_eta,max_eta,num_eta)
scales = 10.0.^linspace(min_log_scale,max_log_scale,num_scale)
shapes = linspace(min_shape,max_shape,num_shape)
num_stars = 16000
eta = 0.2
shape = 0.1
scale = 1.0

srand(42)
@time result = eval_model_on_grid_parallel(etas,shapes,scales,num_stars;
    num_evals = num_evals, true_eta = eta, true_shape = shape, true_scale = scale);

PyPlot.contour(log10(scales),shapes,[minimum(result[:,j,k]) for j in 1:num_scale,
    k in 1:num_shape])
plot(log10([scale]),[shape],"ro")  # Put dot where true values of parameters are
xlabel(L"$\log_{10}(\mathrm{scale})$");  ylabel("shape");
#show()


# vim: set sw=2 sts=2 et :
```