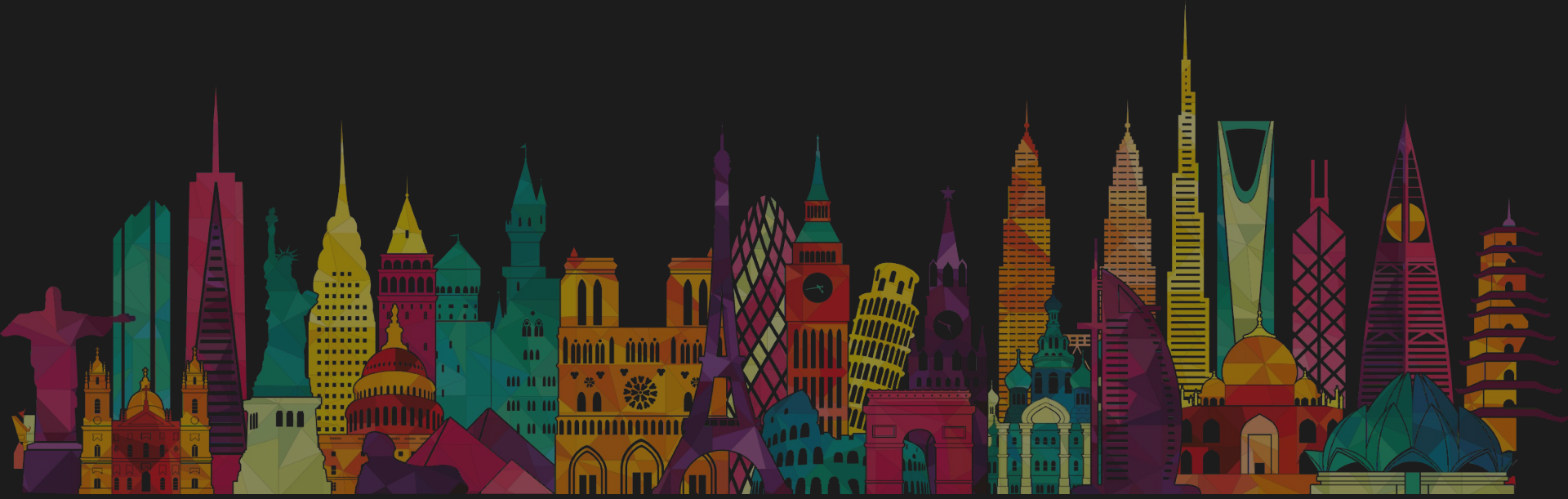
How to not lose a mind by paralelizing a feedback loop?

Feedback loops are notoriously hard to reason about and debug when parallelism is introduced. In this poster, I will show a use case of the TaskMaster for using it to do adaptive and parallel sampling to make your plots faster with constrained resources.



Janis Erdmanis (GitHub: akels, Twitter: graphitewriter, janiserdmanis.org)

An idea worth exploring

Motivation: *Often when we face an embarrassingly parallelizable problem which runs a long time, we reach out for a simple `pmap` command, and if we are lucky, we can run that on the cluster with many cores. However, often it is the case when we want to run our figure faster with resources which are in front of us.*

Another approach is to speed up the calculation by chhosing the grid on which the function is sampled, but that is not alwasy convininet. Instead the grid itslef can adapt as knowldge of evaluated points of the function comes in. Which thus forms a feedback loop.

When paralelism to feedback loop is introduced it produces races and thus for optimal use of resources the result of it is stockhastic making debuggin notoriosuly dificult. Additionally the learning strategies can be executed on different kind of paralizable hardware (CPU, GPU, Cluster jobs, etc.). This is where `TaskMaster ` package comes to save us from misery.

Background: *curreently I am doing a PhD in theorethical condensed matter physics (topological effects in multiterminal superconducting junctions). Started using Julia since `v0.3.3` for my Msc project to get interactivity of Python and speed of C. Quicly got hooked up in reading cool developments on Discourse and later also on Slack.*

Can Julia do better than Python? *It is a common practice that a good craaftsman will make good stuff with bad tool. I do not agree with such sentiment and observed multtiple debates whether tomatous are vegetables or fruits (OOP talk). But the idea was cool and thus started to think how to make Julia API for Python adaptive package my collegues were making.*

Exploring how to solve this problem in Julia was intereseting because:

+ A very natural paralelism abstractions.
+ Channel makes for easy piping
+ Type system and mutliplt dispatch <B
+ Perfomrnace. If you hacve contrained reesource better make susre that f(x) runs as fast as possible.

First let’s abstract a learning strategy

The TaskMaster itself does not implement any learners (appart from making tests). To use TaskMaster one subtypes `AbtractLearner` and adds `ask!` and `tell!` methods which are expected to be determinsitic.

```
using Adaptive
learner = AdaptiveLearner1D((0,1))
```

Which define interval (0,1) at whihch the function is going to be sampled.

Now the learner can be used to `ask!` points.

```
xi =  ask!(learner, si)
```

Where `si` contains an external data passed to the learner, for example, some random number. For Adaptive.jl package one just passes value si=true.

The next step is to evaluate the function and feed it back to the learner

```
yi = f(xi)
tell!(learner, yi)
```

When execution is sequential one can write full execution as a single while loop:

```
while !(learner.loss() < step)
        xi = ask!(learner,true)
        yi = f(xi)
        tell!(learner,(xi,yi))
End
```

where step is just a convergance parameter specific to a particular learner.

Paralelism on the other hand is harder due to race conditions. For simplicity let’s assume that we have a process which takes values from tasks Channel and evaluates puts results in a form (xi,f(xi)) in the results Channel. Then the evaluation can be written as simple as:

```
unresolved = 0
while true

    if !(learner.loss() < step)

        xi = ask!(learner,true)
        put!(tasks,xi)
        unresolved += 1

        if unresolved < N
            continue
        end
    end

    if unresolved == 0
        break
    end

    yi = take!(results)
    tell!(learner,yi)
    unresolved -= 1

end
```

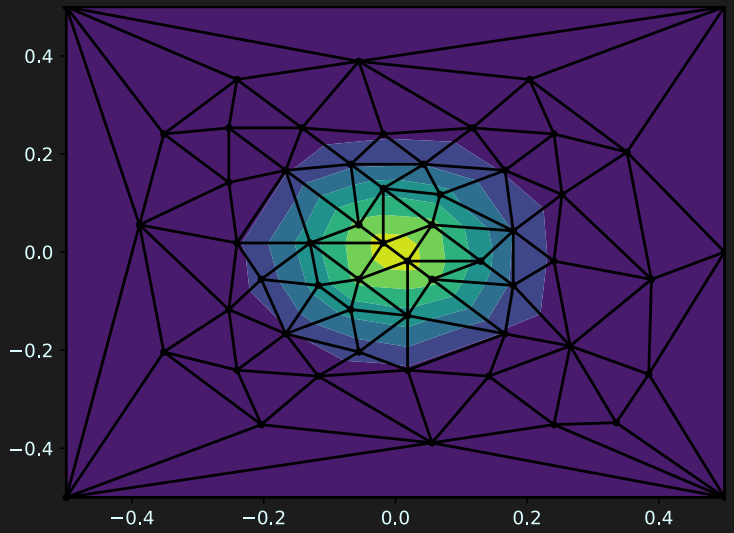
TaskMaster

The idea of TaskMaster is that writing of that while loop accurately counting `unresolved` tasks can be a little challenging for every time one would like to make a figure. Additionally to use the loop one needs to set up evaluation Task between tasks and results Channel.

```
using Adaptive
using TaskMaste

@everywhere f(p) = exp(-p[1]^2 - p[2]^2)

master = WorkMaster(f)
learner2d = AdaptiveLearner2D([(-3,+3),(-3,+3)])
loop = Loop(master,learner2d)
output = evaluate!(loop,learner->learner.loss()<0.05)
```



Here `WorkMaster` sets up processes for evaluteing `f(x)` from input to output Channel. `Loop` defines what is to be evaluated with which learner and `evaluate!` is actually running the loop until specified convergence condition.

Debugging Learner?

Let's imagine a situation where you had spent hours evaluating the function with a Learner. For some particular reason looking at the output, the Learner seems had misbehaved. The question then is how one could debug that?

Through replaying the master as long as learner is deterministic:*

```
hmaster = HistoryMaster(output,length(master.slaves))
hlearner = AdaptiveLearner2D([(-3,+3),(-3,+3)])
hloop = Loop(hmaster,hlearner,loop->println("Learner state $
(hloop.hlearner.state)"))
evaluate!(hloop,learner->learner.loss()<0.05)
```

Thus one can understand the learner by knowing what state caused the problems.

** Adaptive learners uses random numbers and thus the replay would not work. It would be possible if random number or it's seed would be passed with `ask!` method.*

Conclusions and References

Conclusions:

- Abstractions Julia offers by default almost paralele executor part redundant (TaskMaster) in contrast to Python (Runner module).
- The costs of using Adaptive pacakge is quite heavy when all python dependencies gets installed. Also it is in active development thus the code just might break after a while.
- I showed how deterministic learners can be replayed for debugging their state.
- Using `pmap` is easier as output of it can be plotted and saved with less steps.

Thank You

References:

janiserdmanis.org/TaskMaster.jl

github.com/python-adaptive/adaptive

Bas Nijholt and Joseph Weston and Jorn Hoofwijk and Anton Akhmerov (2019). Adaptive: parallel active learning of mathematical functions. Zenodo