18.337 Final Project:
Nuclear MOC in Julia

Sterling Harper

December 2015

1 Description of the problem

Nuclear reactors operate on a nuclear chain reaction based on neutrons. Free neutrons in the reactor collide with atomic nuclei and occasionally cause those nuclei to fission which in turn produces more neutrons. Naturally, nuclear scientists and engineers spend a great deal of time developing computational methods that model the dynamics of neutrons in reactors. In this project, I will be implementing one these methods—MOC or Method of Characteristics—in Julia.

The geometry of my problem is a "PWR pincell". The first part of that, "PWR" stands for Pressurized Water Reactor. This type of reactor is the most

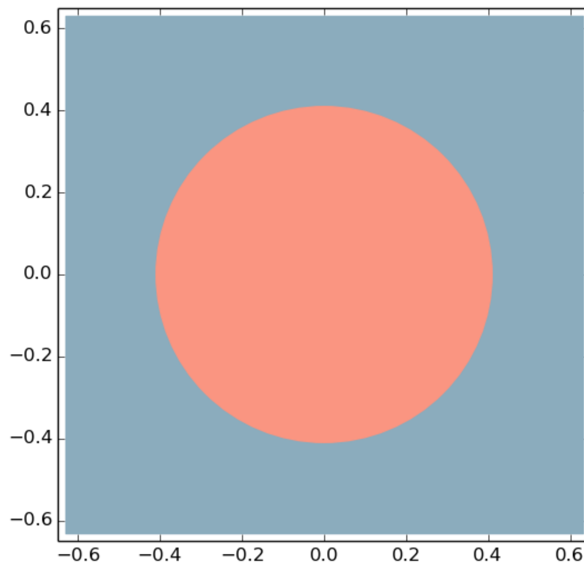

Figure 1: The simulated pincell: a Uranium fuel rod surrounded by water.

popular design for commercial nuclear power plants. The core of the reactor is made up of tens of thousands of long, thin (4m long, 1cm diameter) Uranium rods called fuel rods or fuel pins. We can roughly determine the properties of a PWR by modeling an infinite lattice of fuel pins or a single pin with reflective boundary conditions, i.e. a pincell. The pincell I used in my project is depicted in Figure 1.

2 Method of Characteristics

MOC works by integrating the Boltzmann equation over the characteristic paths traveled by neutrons. Neutrons travel in straight lines between collisions so our characteristics are straight rays that we call tracks. We generally pick an even number of equally spaced azimuthal angles to point the tracks in. A scheme with 64 azimuthal angles is shown in Figure 2. We then repeat all of those tracks with some given spacing. Figure 3 shows the repetitions for a single azimuthal angle with a 5 mm spacing.

The MOC algorithm is shown in Algorithm 1. The details of the physics are unimportant for this project, but it useful to highlight a few key points about that algorithm: First, within an iteration n, each track k is independent. In other words, we do not need knowledge about any other track to compute the effects of k so we can integrate over them on different processors without communication until the end of the iteration.

However, across two iterations a track k will have an effect on track k' through the reflective boundary condition (see line 16 in Algorithm 1) and the source term (lines 8, 23, and 25). So these values must be communicated.

A parallel scheme was implemented in Julia with the following constructs: First, lines 3 through 20 of Algorithm 1 were implemented in a "sweep" function. This function returns fluxes ($\Phi_{i,g}$) and boundary currents ($\psi_{k,g,p}$) in a "SweepResult" object.

```
@everywhere type SweepResult
    next_flux::Array{Float64}
    next_current::Dict
end
```

A "merge!" function for "SweepResult"s are implemented as follows:

```
function merge!(a::SweepResult, b::SweepResult)
    a.next_flux += b.next_flux
    a.next_current = merge(a.next_current, b.next_current)
    a
end
```

With that framework established, parallelizing across two processors can be done with these short statements (input arguments removed for clarity):

```
p1_ref = @spawn sweep(...)
```

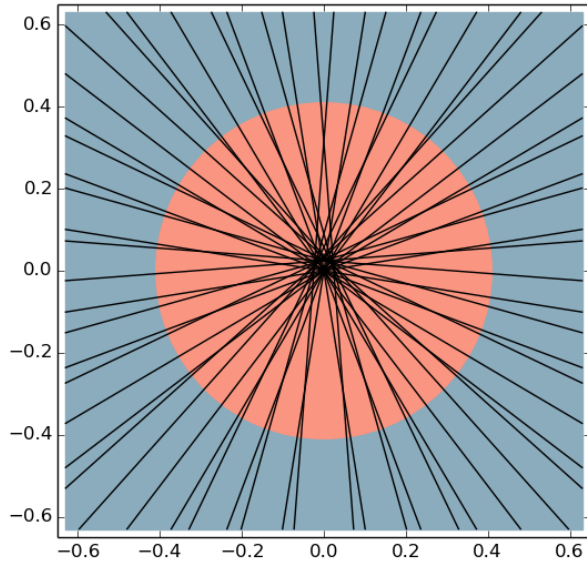

Figure 2: MOC tracks distributed over 64 azimuthal angles.

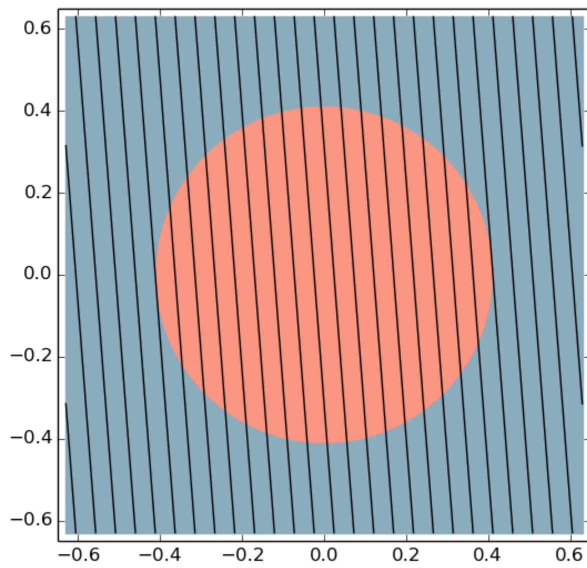

Figure 3: MOC tracks with a 5 mm spacing

Algorithm 1 The MOC loop

```
1: while Not converged do
2:     Iteration n ← n + 1
3:     for Azimuthal angle m do
4:         for Track k ∈ m do
5:             for Segment s ∈ k do
6:                 for Energy group g do
7:                     for Polar angle p do
8:                         Δψ ← (ψ^n_{k,g,p} − Q^n_{i,g}/Σ^t_{i,g})(1 − e^{−τ_{k,g,p}})
9:                         Φ_{i,g} ← Φ_{i,g} + (4π/A_i) ω_{m,p,k} sin(θ_p) Δψ
10:                        ψ^n_{k,g,p} ← ψ^n_{k,g,p} − Δψ_{i,g}
11:                    end for
12:                end for
13:            end for
14:            for Energy group g do
15:                for Polar angle p do
16:                    ψ^{(n+1)}_{k',g,p} ← ψ^n_{k,g,p}
17:                end for
18:            end for
19:        end for
20:    end for
21:    for Region i do
22:        for Energy group g do
23:            Q^{(n+1)}_{i,g} ← (1/k) χ ν Σ^F_{i,g} Φ_{i,g}
24:            for Energy group g' do
25:                Q^{(n+1)}_{i,g} ← Q^{(n+1)}_{i,g} + Σ^S_{i,g'→g} Φ_{i,g'}
26:            end for
27:        end for
28:    end for
29: end while
```

```
p2_ref = @spawn sweep(...)
result = fetch(p1_ref)
result = merge!(result, fetch(p2_ref))
```

3 Testing

In this section, the Julia MOC implementation is compared with a Python and a C++ implementation. The Python code is a simple solver that I wrote and it closely matches the Julia code. The C++ code is a mature research code called OpenMOC. It has many features beyond the Python and Julia implementations, but it will still serve as an interesting point of comparison.

The speed of various Julia implementations are shown in Table 1. The Python program runs a distressingly 41× slower than the C++ code. Simply translating the code directly into Julia dropped the runtime down to 5×. Some small serial optimizations (reusing data structures rather than allocating new ones) dropped that down to 3×. Unfortunately, the parallel implementation performed worse than the serial versions at 16×.

4 Discussion

I think Julia performed very well in this short experiment. Simply translating my Python code into Julia gave me an order of magnitude performance boost. With relatively little effort, I was able to get my Julia code running only 3× more slowly than a mature code with very low-level optimizations (like a linear interpolation table for evaluating exponentials). And I believe the Julia code is very readable—more readable than the C++ code and on par with Python despite the unfamiliar notion of non-type-bound procedures. I had to write a lot of data I/O code not mentioned in this write-up, and I found Julia's implementation of regex and object constructors to be friendly enough.

Obviously my failure to get a parallel speed-up was disappointing. That said, I don't really blame Julia. I know that passing dictionaries around is likely very inefficient, and there is also a way to cleverly divide up the work so that the current dictionaries do not need to be communicated at all. In other

Program	Runtime
C++	2.8 s (1×)
Python	117.0s (41×)
Julia, direct translation	14.4 s (5×)
Julia, serial optimization	9.4 s (3×)
Julia, 2 processors	46.1 s (16×)

Table 1: Runtime of an MOC test problem with the two reference codes and the three Julia implementations. The parentheses show by what factor the runtime is larger than the C++ implementation.

words, I need to do some real computer science to get the parallel efficiency. I had secretly hoped that I could replace real computer science with a choice of language, but alas, parallel computing is just too complex.

I will definitely consider using Julia in the future for my small software projects where Python was previously my go-to language. It's compact, readable, has a healthy ecosystem of packages, and it's 10$\times$ faster than Python. I'm impressed.