Astro 585: HW 3

Codename: The Maxwell-Jüttner Distribution

February 6, 2014

1 Common Function Benchmarks

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

2 File I/O

First the program file, then the output:

```
#!/usr/bin/env julia
#!/Applications/Julia.app/Contents/Resources/julia/bin/julia
# yeah, haven't added julia to my path on mac

# 2a
srand(42)
v = rand(2^10)

time = @elapsed println(v)
println("# 2a")
println("Print to stdout takes ", time, " seconds")

# 2b,c
println("\n# 2b,c")
time = @elapsed writedlm("torvalds.dat", v, '\t')
println("Writing: ", time, " seconds")
time = @elapsed readdlm("torvalds.dat", '\t', Float64)
println("Reading: ", time, " seconds")
time = @elapsed readdlm("torvalds.dat", '\t', Float64)
println("Reading: ", time, " seconds")
time = @elapsed readdlm("torvalds.dat", '\t', Float64)
println("Reading: ", time, " seconds")
println("\nWriting to a file is about 10 times faster than to writing to
stdout. Initial reading is slow, but subsequent ones are amazingly fast,
probably because julia (and the OS) caches the read.\n")
```

```
v = rand(2^20)
time = @elapsed writedlm("torvalds.dat", v, '\t')
println("Writing2^20: ", time, " seconds")
time = @elapsed readdlm("torvalds.dat", '\t', Float64)
println("Reading2^20: ", time, " seconds")
time = @elapsed readdlm("torvalds.dat", '\t', Float64)
println("Reading2^20: ", time, " seconds")
time = @elapsed readdlm("torvalds.dat", '\t', Float64)
println("Reading2^20: ", time, " seconds")
println("\nYeah, cache ain't big enough no more.")

# 2d
println("\n# 2d")
function writebin(filename::String, v)
    # ignore error handling
    stream = open(filename, "w")
    write(stream, v)
    close(stream)
end
function readbin!(filename::String, v)
    stream = open(filename, "r")
    v[:] = read(stream, typeof(v[0]), length(v))
    close(stream)
    return v
end

time = @elapsed writebin("wozniak.bin", v)
println("Writing2^20 in binary: ", time, " seconds")
time = @elapsed readbin!("wozniak.bin", v)
println("Reading2^20 in binary: ", time, " seconds")
time = @elapsed readbin!("wozniak.bin", v)
println("Reading2^20 in binary: ", time, " seconds")
time = @elapsed readbin!("wozniak.bin", v)
println("Reading2^20 in binary: ", time, " seconds")

println("\nASCII file size: ", stat("torvalds.dat").size / 2^20, " MB")
println("Binary file size: ", stat("wozniak.bin").size / 2^20, " MB")

println("\nSmaller, meaner, faster!")

# 2e
println("\n# 2e")
using HDF5, JLD
time = @elapsed @save "holy.jld" v
println("Writing2^20 in hdf5: ", time, " seconds")
```

```
time = @elapsed @save "holy.jld" v
println("Writing2^20 in hdf5: ", time, " seconds")
time = @elapsed @load "holy.jld"
println("Reading2^20 in hdf5: ", time, " seconds")
time = @elapsed @load "holy.jld"
println("Reading2^20 in hdf5: ", time, " seconds")
time = @elapsed @load "holy.jld"
println("Reading2^20 in hdf5: ", time, " seconds")

println("\nHDF5 file size: ", stat("holy.jld").size / 2^20, " MB")

println("\nWriting is slower than pure binary, but reading is faster, which
seems odd. Possibly the 'read()' call does some buffer management that the HDF5
library can avoid with more information.")

# 2f
println("\n# 2f")
println("The ASCII format will have to read all elements, because each element
has a different length, and we need to find the delimiters. The pure binary
format will probably be fastest, since I would just calculate the position of
the relevant element (i * sizeof(Float64)) and then seek to it. HDF5 might be
as fast as the pure binary format, but I imagine that it would first read the
beginning of the file to find out the structure of the file, and then seek to
the relevant position.")

# 2g
println("\n# 2g")
println("ASCII is human readable, and human editable. However, the simple
version used in this homework I would never use in any serious work, as it is
too easy to confuse what each number means. ASCII in general is not useful for
storing large amounts of data, as it takes more than twice as much space, and
is significantly slower. It may also loose some precision when converting to a
decimal representation. However, it may be useful for small data tables.

Using a pure binary file is simple, but not so great, since the precise format
is easily forgotten, and may change depending on which machine you run it.

HDF5 seems nice for large data sets, but as for the pure binary format you will
need a program to read and edit it, so I would not use it for configuration or
parameter files. It is likely precisely enough defined to be the same no matter
what machine you run your program on.")

# 2h
# Interesting stuff first...
println("\n# 2h")
```

```
using YAML
println("The file 'knuth.yaml' was created with './2_makeyaml.py > knuth.yaml',
because yaml for julia cannot yet write a YAML file at this time.")
time = @elapsed YAML.load(open("knuth.yaml"))
println("Reading2^10 in YAML: ", time, " seconds")
time = @elapsed YAML.load(open("knuth.yaml"))
println("Reading2^10 in YAML: ", time, " seconds")

println("\nAt this time, YAML is not a serious contender for storing any amount
of data. Besides the first read time being, uhm, slow, the table is not really
human readable. However, it looks like a really nice configuration file
format.")
```

And for creating a YAML file I used the file `2_makeyaml.py`:

```
#!/usr/bin/env python

import yaml
import numpy as np

v = np.random.rand(2**10)

# Convert to list so that we don't get random binary python specific yaml output
# (Odd that the following doesn't work, there are still numpy specific types in
# there: v = list(v))
w = []
for i in v:
    w += [float(i)]

print yaml.dump({'random vector': w})

#x = yaml.load(open("knuth.yaml"))
#print x['random vector']
```

The output is:

```
<snip random vector fo 1024 elements>

# 2a
Print to stdout takes 0.270311566 seconds

# 2b,c
Writing: 0.101148976 seconds
Reading: 3.015733168 seconds
Reading: 0.001776553 seconds
Reading: 0.001718864 seconds
```

```
Writing to a file is about 10 times faster than to writing to
stdout. Initial reading is slow, but subsequent ones are amazingly fast,
probably because julia (and the OS) caches the read.

Writing2^20: 1.245572694 seconds
Reading2^20: 1.769823281 seconds
Reading2^20: 1.74360053 seconds
Reading2^20: 1.744750048 seconds

Yeah, cache ain't big enough no more.

# 2d
Writing2^20 in binary: 0.275027949 seconds
Reading2^20 in binary: 0.11283971 seconds
Reading2^20 in binary: 0.02341513 seconds
Reading2^20 in binary: 0.023484832 seconds

ASCII file size: 18.268983840942383 MB
Binary file size: 8.0 MB

Smaller, meaner, faster!

# 2e
Writing2^20 in hdf5: 0.816217353 seconds
Writing2^20 in hdf5: 0.234449809 seconds
Reading2^20 in hdf5: 0.074714466 seconds
Reading2^20 in hdf5: 0.012444531 seconds
Reading2^20 in hdf5: 0.011976035 seconds

HDF5 file size: 8.002532958984375 MB

Writing is slower than pure binary, but reading is faster, which
seems odd. Possibly the 'read()' call does some buffer management that the HDF5
library can avoid with more information.

# 2f
The ASCII format will have to read all elements, because each element
has a different length, and we need to find the delimiters. The pure binary
format will probably be fastest, since I would just calculate the position of
the relevant element (i * sizeof(Float64)) and then seek to it. HDF5 might be
as fast as the pure binary format, but I imagine that it would first read the
beginning of the file to find out the structure of the file, and then seek to
the relevant position.

# 2g
```

```
ASCII is human readable, and human editable. However, the simple
version used in this homework I would never use in any serious work, as it is
too easy to confuse what each number means. ASCII in general is not useful for
storing large amounts of data, as it takes more than twice as much space, and
is significantly slower. It may also loose some precision when converting to a
decimal representation. However, it may be useful for small data tables.

Using a pure binary file is simple, but not so great, since the precise format
is easily forgotten, and may change depending on which machine you run it.

HDF5 seems nice for large data sets, but as for the pure binary format you will
need a program to read and edit it, so I would not use it for configuration or
parameter files. It is likely precisely enough defined to be the same no matter
what machine you run your program on.

# 2h
The file 'knuth.yaml' was created with './2_makeyaml.py > knuth.yaml',
because yaml for julia cannot yet write a YAML file at this time.
Reading2^10 in YAML: 9.325846978 seconds
Reading2^10 in YAML: 0.334877953 seconds

At this time, YAML is not a serious contender for storing any amount
of data. Besides the first read time being, uhm, slow, the table is not really
human readable. However, it looks like a really nice configuration file
format.
```

3 Testing

First the program file, then the output:

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

require("2_io.jl")
using Base.Test
using HDF5, JLD

# Test different numbers on each run. Be sure to provide the computer with
# entropy, e.g. by moving the mouse.
println("Get entropy...")
srand("/dev/random")
v = rand(2^10)

# test_rw():
#    - This functions tests whether the function 'readfn' reads the data
#      written by 'writefn' faithfully. That is, is it the same, and is it
#      represented the same?
```

```
#    - 'v' is any data used for the testing, typically a vector of floats.
#    - 'filename' is the name of a file to write to. It will be left on the
#      disk.
#    - 'approx' determines if small round-off errors are OK.
function test_rw(writefn, readfn, v, filename; approx=false)
    # make absolutely sure we don't just overwrite the array
    vcopy = deepcopy(v)
    writefn(filename, v)
    vread = readfn(filename, vcopy)
    if approx == true
        @test_approx_eq vread v
    else
        @test vread == v
    end
end


function test_iofn(writefn, readfn, v, filename; approx=false, handle_scalars=false)
    shortfn(x) = test_rw(writefn, readfn, x, filename, approx=approx)
    shortfn(v)
    test_rw(writefn, readfn, v, filename, approx=approx)
    if handle_scalars == true
        test_rw(writefn, readfn, 1.0, filename, approx=approx)
    else
        @test_throws test_rw(writefn, readfn, 1.0, filename, approx=approx)
    end
    @test_throws test_rw(writefn, readfn, None, filename, approx=approx)
end

asciiwritefn(filename, v) = writedlm(filename, v, '\t')
asciireadfn(filename, v) = readdlm(filename, '\t', Float64)

# 3a
println("# 3a")
test_iofn(asciiwritefn, asciireadfn, v, "torvalds.dat", approx=true, handle_scalars=true)
println("ASCII success! (with approximation)")

test_iofn(writebin, readbin!, v, "wozniak.bin", approx=false)
println("Binary success! (exact)")

hdf5writefn(filename, v) = @save filename v
function hdf5readfn(filename::String, v)
    using HDF5, JLD
    # The following line doesn't like me. Why? Because the function
    # 'jdlopen(Symbol)', which is called inside '@load', does not exist. It
```

```
    # seems to be something about the fact that the argument to 'jdlopen()' is
    # a 'Symbol', but I am not sure why this would suddenly make a difference.
    # I already said that 'filename' is a String?
    @load filename
end

test_iofn(hdf5writefn, hdf5readfn, v, "holy.jld")
println("HDF5 success! (exact)")
```

With output:

```
Get entropy...
# 3a
ASCII success! (with approximation)
Binary success! (exact)
ERROR: no method jldopen(Symbol)
while loading /home/hsgg/pennstate/585-comp/hw3/3_test.jl, in expression starting on line 58
```

4 Documentation

I added some documentation to the function 'test_rw()' in part 3 of the homework.