Parallel & Distibuted Computing: Lecture 20

Alberto Paoluzzi

November 18, 2019

Program C for individual projects

1. Introduction to molecular dynamics
2. Synthesis of chapters 5-6 of “Julia High Performance” book
3. Fast Numbers
4. Using Arrays

Introduction to molecular dynamics

Introduction

Molecular dynamics (MD) is a computer simulation method for analyzing the physical movements of atoms and molecules. The atoms and molecules are allowed to interact for a fixed period of time, giving a view of the dynamic "evolution" of the system. (Wikipedia)

Project material

Molecular Dynamics Simulations with Julia
Verlet integration
https://github.com/ejc44/MD
Prof. Edelman MIT-18.337/6.338 – Modern Numerical Computing with Julia

The Project Report (Emily Crabb, MIT)

This project consists of one serial and several parallel versions of a molecular dynamics simulation in the Julia programming language.

These implementations are contained in IJulia notebooks.

The entire project is publicly available on Github at:
https://github.com/ejc44/MD.

Specific details regarding each version are included in each notebook, and separate timing notebooks are provided that use the BenchmarkTools.jl package.

Clone the repository

$ git clone https://github.com/ejc44/MD

Using Notebooks with IJulia

Installing notebook reader IJulia

```
julia> using Pkg

julia> Pkg.add("IJulia")
```

Running the IJulia Notebook

```
julia> using IJulia
julia> notebook()
```

Give a look to https://github.com/ejc44/MD

MD

Molcular dyanmics implementation for CS 6.338 Final Project

This repo includes a serial and several parallel versions of a molecular dynamics simulation in the Julia programming language contained in Julia notebooks. Notable features include

1. The option to read parameters (like number of simulation steps) and/or initial data (starting configuration: position, velocity, acceleration, forces) from external files. These files must be in the same folder as this notebook and have the correct names (as specified in the code).
2. The option to directly specify the parameters in the notebook. Note: These parameters are all constants, so one must restart the kernel to redefine them.
3. The option to save the parameters and output data to external files.
4. The option to model finite and infinite systems.
5. The ability to make finite systems periodic or non-periodic.
6. If the initial data / configuration is not specified in a file, it can be generated in the code. For example, the starting positions are random within the specified box size. The user can modify any of the initial conditions by altering the initialize() function.

Synthesis of chapters 5-6 of “Julia High Performance” book

Fast Numbers

1. Numbers in Julia, Their layout and storage
2. Trading performance for accuracy
3. Subnormal numbers

Using Arrays

1. Arrays internals in Julia
2. Bounds checking
3. Allocations and in-place operations
4. Broadcasting
5. Array views
6. SIMD parallelization
7. Specialized array types
8. Writing generic library functions with arrays

Fast Numbers

Numbers in Julia, Their layout and storage

WORD_SIZE

Integers

```
julia> bitstring(3)
"0000000000000000000000000000000000000000000000000000000000000011"

julia> bitstring(-3)
"1111111111111111111111111111111111111111111111111111111111111101"

julia> isbitstype(Int64)
true

julia> isbitstype(String)
false
```

Advantages

```
julia> myadd(x, y) = x + y
myadd (generic function with 1 method)

julia> @code_native myadd(1,2)
```

Numbers in Julia, Their layout and storage

WORD_SIZE

```
julia> typemax(Int64)
9223372036854775807

julia> bitstring(typemax(Int32))
"01111111111111111111111111111111"

julia> typemin(Int64)
-9223372036854775808

julia> bitstring(typemin(Int32))
"10000000000000000000000000000000"
```

Integer overflow

```
julia> 9223372036854775806 + 1
9223372036854775807

julia> 9223372036854775806 + 1 + 1
-9223372036854775808
```

Numbers in Julia, Their layout and storage

BigInt

```
julia> big(9223372036854775806) + 1 + 1
9223372036854775808

julia> big(2)^64
18446744073709551616
```

Numbers in Julia, Their layout and storage
The Floating Point

the Institute of Electrical and Electronics Engineers (IEEE) 754 binary standard

The binary storage standard for the 64-bit floating point numbers consists of 1 sign bit, 11 bits of exponent, and 52 bits of the mantissa (or the significand).

```
julia> bitstring(2.5)
"0100000000000100000000000000000000000000000000000000000000000000"

julia> bitstring(-2.5)
"1100000000000100000000000000000000000000000000000000000000000000"

julia> function floatbits(x::Float64)
           b = bitstring(x)
           b[1:1]*"|"*b[2:12]*"|"*b[13:end]
       end
floatbits (generic function with 1 method)

julia> floatbits(2.5)
"0|10000000000|0100000000000000000000000000000000000000000000000000"
```

Numbers in Julia, Their layout and storage

FastMath

julia> function sum_diff(x) n = length(x); d = 1/(n-1) s = zero(eltype(x)) s = s + (x[2] - x[1]) / d for i = 2:length(x)-1 s = s + (x[i+1] - x[i+1]) / (2*d) end s = s + (x[n] - x[n-1])/d end sum_diff (generic function with 1 method)

julia> function sum_diff_fast(x) n=length(x); d = 1/(n-1) s = zero(eltype(x)) @fastmath s = s + (x[2] - x[1]) / d @fastmath for i = 2:n-1 s = s + (x[i+1] - x[i+1]) / (2*d) end @fastmath s = s + (x[n] - x[n-1])/d end sum_diff_fast (generic function with 1 method)

julia> t=rand(2000);

julia> @btime sum_diff($t)

4.684 μs (0 allocations: 0 bytes) -332.05573916220476

julia> @btime sum_diff_fast($t) 585.250 ns (0 allocations: 0 bytes) -332.05573916220465

julia> macroexpand(Main, :(@fastmath a + b / c)) :((Base.FastMath).add_fast(a, (Base.FastMath).div_fast(b, c)))

Trading performance for accuracy

aaaaaa

Subnormal numbers

aaaaaa

Using Arrays

Arrays internals in Julia

aaaaaa

Bounds checking

aaaaaa

Allocations and in-place operations

aaaaaa

Broadcasting

aaaaaa

Array views

aaaaaa

SIMD parallelization

aaaaaa

Specialized array types

aaaaaa

Writing generic library functions with arrays

aaaaaa