SKILLPILLS

Skill Pill: Julia

Lecture 4: Distributed and parallel computing

Valentin Churavy James Schloss

Okinawa Institute of Science and Technology
valentin.churavy@oist.jp
james.schloss@oist.jp

July 13, 2017

OIST

1. Levels of parallelism
2. Instruction level parallelism
3. Threading
4. Distributed

Introduction

Introduction

Necessary packages

- SIMD.jl
- MPI.jl
- DistributedArrays.jl
- CUDAnative.jl if your computer has a NVidia GPU

Levels of parallelism

- Instruction level parallelism
- Shared-memory and threading
- Distributed
- Accelerators e.g.GPGPU

What is instruction level parallelism

```julia
function padd(a, b, x, y)
  c = a + b
  z = x + y
  return c, z
end
```

Observation

The two operations are independent of each other and we could execute them in parallel.

1. Use `@code_llvm` and `@code_native` to understand what is happening
2. Establish a baseline performance with `@benchmark`
3. Start Julia with `julia -O3`
4. Compare the llvm and native code and your benchmark results
5. Note that there is next to no performance benefit in this example, but that changes once you scale up

SIMD and loops

```
function add(out, x, y)
  for i in 1:length(out)
    out[i] = x[i] + y[i]
  end
end
```

Observation

Each loop iteration is independent.

1. Learn about `@inbounds`

Reductions and loop-dependencies

```
function sum(x)
  acc = 0.0
  for i in 1:length(x)
     acc += x[i]
  end
end
```

Observation

Is each loop iteration independent from each other? Yes and no. Standard addition is associative and the order of operations has no impact. Floating-point addition is non-associative and the order of operations is important. The compiler cannot vectorise this loop, without changing the semantics.

1. Learn about `@simd`

Explicit SIMD

SIMD.jl

Instead of relying on the compiler to optimise and vectorise our code correctly we can also write explicit SIMD code.

```julia
using SIMD
function add(out::Vector{Float64}, x::Vector{Float64},
    y::Vector{Float64})
  # My laptop supports AVX 256bit 4xFloat64
  @assert length(x) % 4 == 0
  for i in 1:4:length(x)
    vx = vload(Vec{4, Float64}, x, i)
    vy = vload(Vec{4, Float64}, y, i)
    vo = vx + vy
    vstore(vo, out, i)
  end
end
```

Explain current fork join model and caveats

Simple example

complex example with loop splitting, random etc...

Atomics

MPI

DistibutedArrays