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

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

```
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
```

Fork-Join model

Caveat

Julia threading model is based on a fork-join approach and is still considered experimental. The Julia developers are working with Intel on bringing a modern and full-fledged threading model to Julia for 1.0

Fork-Join model

Caveat

Julia threading model is based on a fork-join approach and is still considered experimental. The Julia developers are working with Intel on bringing a modern and full-fledged threading model to Julia for 1.0

What is fork-join

Fork-join describes the control flow that a group of threads undergo. Execution is forked across to all threads and at a later point execution is joined together. In Julia all threads have to execute the same lambda.

Start Julia with `JULIA_NUM_THREADS=4 julia` the number of threads needs to be set before starting Julia.

```
using Base.Threads

a = zeros(nthreads()*10)
@threads for i in 1:length(a)
  a[i] = threadid()
end
```

Caveats

Special care needs to be taken for access to global state, which includes IO and random numbers.

Further examples

```
A = zeros(10_000)
@threads for i in 1:length(A)
  A[i] = rand() # Note rand uses a global variable
      Base.GLOBAL_RNG
end

rngs = [MersenneTwister(rand(UInt64)) for i in 1:nthreads()]
@threads for i in 1:length(A)
  A[i] = rand(rngs[threadid()])
end
```

Support for Atomics

```
acc = 0
@threads for i in 1:10_000
  acc += 1
end
```

To prevent the datarace we can use Atomics to ensure read-write consistency.

```
acc = Atomic{Int64}(0)
@threads for i in 1:10_000
  atomic_add!(acc, 1)
end
```

MPI

Julia supports MPI through `MPI.jl`. If you are comfortable with MPI you can use it with very low overhead.

Parallel programming

Starting Julia with `julia -p 4` will create one master process and four worker processes. You can use remotecalls to execute work on a worker and to get back the results. Based on this `DistributeArrays.jl` provides a powerful global array interface. `@everywhere` will execute a piece of code on all processes.

CUDAnative

CUDAnative

A recent development is the ability to write raw CUDA code in Julia!
Learn more about it at `https://github.com/JuliaGPU/CUDAnative.jl`