HPS

http://davidemattioli.net

Davide Mattioli
Supervisor: Tino Meisel

Julia Application in HPC and Scientific Computing

An Overview of Performance and Capabilities

Table of Contents

Outline

What is Julia?

“We are greedy: we want more. We want the speed of C with the dynamism of Ruby. We want a language with a familiar mathematical notation like Matlab. We want something as usable for general programming as Python, as easy for statistics as R. We want it interactive and we want it compiled.”

— Jeff Bezanson, Stefan Karpinski, Viral B. Shah, Alan Edelman

History of Julia

- Released in 2012
- Developed to combine the performance of C with the ease of use of Python
- Over 1,000 contributors and a growing ecosystem

Core Features

- Optional typing and multiple dispatch
- Just-in-time (JIT) compilation
- High-performance for numerical and computational tasks

Why Julia for HPC?

- High-level syntax similar to Python or Matlab
- Performance comparable to C/C++
- Extensive library support for various scientific computing tasks

Outline

What is Multiple Dispatch?

- **Definition:** Multiple dispatch allows the selection of method to execute based on the types of multiple arguments.
- **Comparison:** Extends beyond single dispatch (common in OOP languages) where only one argument type determines the method.
- **Benefit:** Facilitates more dynamic and flexible code.

Basic Example

- Julia's generic function example:

  `f(x) = 2x`$^2 + x$
- Can be applied to various types without modification:
  - `f(2)` $\rightarrow$ 10
  - `f(2.5)` $\rightarrow$ 18.75
  - `f(1 + 2im)` $\rightarrow -1 + 6im$

The problem

Julia

```julia
#Consider the following function

julia> g(x) = 2*x^2 + x/2
julia> g(1.0)
2.5
#but when we call the function on 1 as an Integer
julia> g(1)
2
#What happened? Well the division operator `/`
#in Julia computes the *exact division*.
julia> g(n::Integer) = 2*n^2 + div(n,2)
g (generic function with 2 methods)
#Now if we evaluate `g` on an `Int`, it returns an `Int` as we wanted
julia> g(1)
2.5
```

Defining Multiple Methods

- Example with specific types:

  `f(x::Int) f(x::Float64) f(x::Any)`

- Calling `f(1)`, `f(1.5)`, and `f("two")`:
  - `f(1)` $\rightarrow$ "This is an Int: 1"
  - `f(1.5)` $\rightarrow$ "This is a Float: 1.5"
  - `f("two")` $\rightarrow$ "This is a generic fallback"

Importance in HPC

- **Performance:** Allows optimized methods for specific data types, leading to faster computations.
- **Flexibility:** Easily extend existing functions for new types without modifying existing code.
- **Parallelization:** Simplifies parallel algorithms by dispatching different methods based on data type.

Outline

Introduction

- Multi-threaded parallelism is essential for exploiting multiple processor cores.
- Julia v1.3.0 introduced a new threading interface for general task parallelism.
- Inspired by parallel programming systems like Cilk, Intel TBB, and Go.

Parallelism in Julia

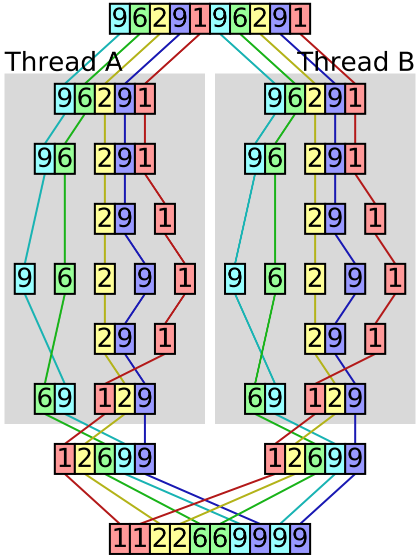
- Any piece of a program can be marked for parallel execution.
- A dynamic scheduler automatically manages task execution on available threads.
- Similar to garbage collection: you spawn tasks without worrying about where they run.

Features of Task Parallelism

- Portable and free from low-level details.
- No need to explicitly start and stop threads or know the number of processors.
- Nestable and composable: Start parallel tasks that call other parallel tasks.
- Significant speedup in computational tasks with multi-threading.
- Flexibility to extend parallel capabilities to the entire Julia package ecosystem.

Example: Parallel Merge Sort

```julia
function psort!(v, lo::Int=1, hi::Int=length(v))
    if lo >= hi
        return v
    end
    mid = (lo + hi) >>> 1
    half = @spawn psort!(v, lo, mid)
    psort!(v, mid+1, hi)
    wait(half)
    merge!(v, lo, mid, hi)
    return v
end
```


9 6 2 9 1 9 6 2 9 1
Thread A
Thread B
9 6 2 9 1
9 6 2 9 1
9 6 2 9 1
2 9 1
9 6 2 9 1
2 9 1
9 6 2 9 1
2 9
6 9 1 2 9
1 2 6 9 9
1 1 2 2 6 6 9 9 9 9

Outline

CUDA.jl: Overview

- CUDA.jl is a Julia package for programming NVIDIA GPUs.
- Provides high-level abstractions and low-level API access.
- Integrates seamlessly with Julia's language features, including metaprogramming.
- Enables writing GPU kernels in pure Julia code.

Key Features of CUDA.jl

- Automatic memory management and garbage collection.
- Support for Julia’s native array types and operations.
- High-level wrappers for CUDA runtime and driver APIs.
- Compilation of Julia code to PTX (Parallel Thread Execution) code.

Defining a GPU Kernel in Julia

```julia
using CUDA

function gpu_addition_kernel(a, b, c)
    i = threadIdx().x
    c[i] = a[i] + b[i]
    return
end
```

- Define GPU kernel using standard Julia functions.
- Utilize CUDA-specific functions like `threadIdx()`.

Executing a GPU Kernel

```julia
# Allocate and initialize arrays
a = CUDA.fill(1.0f0, 1000)
b = CUDA.fill(2.0f0, 1000)
c = CUDA.fill(0.0f0, 1000)

# Launch the kernel
@cuda threads=1000 gpu_addition_kernel(a, b, c)
```

- Allocate arrays on the GPU.
- Use @cuda macro to launch the kernel with specified number of threads.

Advantages of Using CUDA.jl

- Simplifies GPU programming by using Julia’s high-level syntax.
- Reduces boilerplate code for managing GPU resources.
- Enables rapid prototyping and testing of GPU algorithms.
- Integrates with Julia’s ecosystem for data science and machine learning.

Introduction to Multi-GPU Support

- Leveraging multiple GPUs can significantly improve performance.
- Julia provides abstractions to manage multiple GPUs.
- Essential for large-scale scientific computations and deep learning tasks.
- High-level abstractions simplify the use of multiple GPUs.
- Write code that scales across several GPUs with minimal changes.

Example: Multi-GPU Vector Addition

```julia
# Select GPU devices
dev1 = CUDA.Device(1)
dev2 = CUDA.Device(2)

# Allocate arrays on different GPUs
a = CUDA.fill(1.0f0, (1000,), device=dev1)
b = CUDA.fill(2.0f0, (1000,), device=dev2)
c = CUDA.fill(0.0f0, (1000,), device=dev1)

# Define a kernel for vector addition
@target ptx function vadd(a, b, c)
    i = blockIdx().x * blockDim().x + threadIdx().x
    c[i] = a[i] + b[i]
end
```

Comparison with Other Languages

- **Python with TensorFlow/PyTorch:** High-level APIs but requires significant boilerplate for multi-GPU support.
- **C++ with CUDA:** High performance but complex and low-level programming.
- **Julia:** Combines high-level syntax with performance close to C++.
- **R:** Limited support for multi-GPU computing.

Outline

Introduction

- Oceananigans.jl is a fast and user-friendly fluid dynamics package.
- Designed for finite volume simulations of nonhydrostatic and hydrostatic Boussinesq equations.
- Supports both CPU and GPU execution for enhanced performance.
- Developed by the Climate Modeling Alliance and external collaborators.

Impact of Ocean Waves on Weather and Climate Patterns

- **Weather Systems:** Influence atmospheric pressure and wind patterns. Play a role in the development and intensity of storms, hurricanes, and typhoons.
- **Climate Events:** Phenomena like El Niño and La Niña driven by changes in wave patterns and sea surface temperatures. Significant impacts on global weather, agriculture, and water resources.
- **Sea Level Changes:** Understanding wave dynamics is crucial for coastal management and disaster preparedness.

Key Features

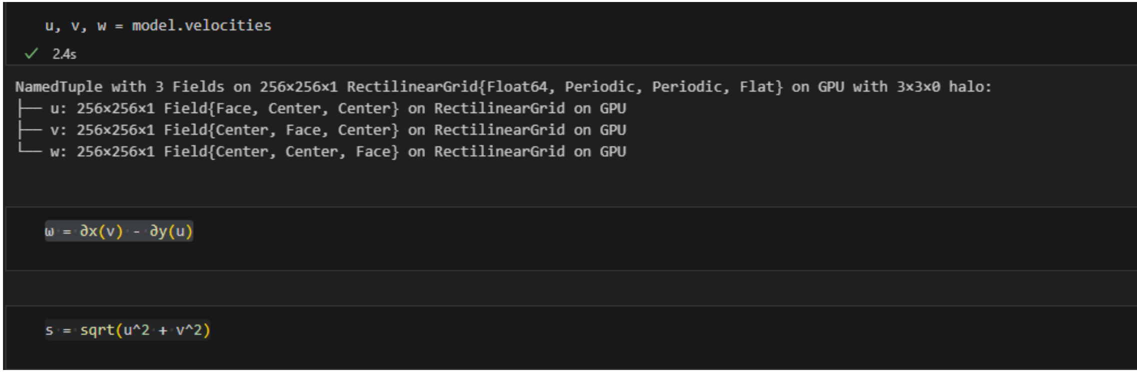
- Performance: Leverages GPU acceleration for fast simulations.
- Flexibility: User-friendly interface for both simple and complex simulations.
- Installation: Easy to install and use with Julia’s package manager.
- Documentation: Comprehensive documentation, examples, and tutorials available.

Example Fluid Turbolence Part Initialisation

```
using Oceananigans

grid = RectilinearGrid(GPU(),size=(256, 256), extent=(2, 2),
                                topology=(Periodic, Periodic, Flat))

model = NonhydrostaticModel(; grid,
                                timestepper = :RungeKutta3,
                                advection = UpwindBiasedFifthOrder(),
                                closure = ScalarDiffusivity(=1e-5))
```

```julia
using Statistics

u, v, w = model.velocities

u = rand(size(u)...)
v = rand(size(v)...)

u .-= mean(u)
v .-= mean(v)

set!(model, u=u, v=v)
simulation = Simulation(model, t=0.2, stop_iteration=1000)
```

Grid



```
u, v, w = model.velocities
✓ 2.4s

NamedTuple with 3 Fields on 256×256×1 RectilinearGrid{Float64, Periodic, Periodic, Flat} on GPU with 3×3×0 halo:
├── u: 256×256×1 Field{Face, Center, Center} on RectilinearGrid on GPU
├── v: 256×256×1 Field{Center, Face, Center} on RectilinearGrid on GPU
└── w: 256×256×1 Field{Center, Center, Face} on RectilinearGrid on GPU
```

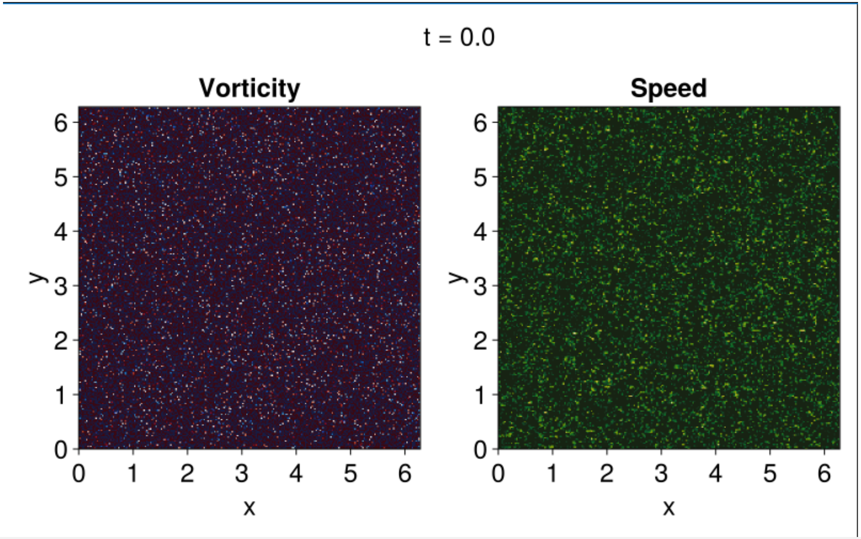
```
ω = ∂x(v) - ∂y(u)
```

```
s = sqrt(u^2 + v^2)
```

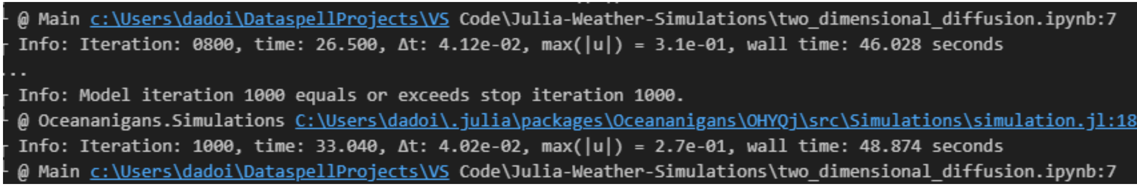
$$\omega = \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}$$

$$s = \sqrt{u^2 + v^2}$$

Initial State



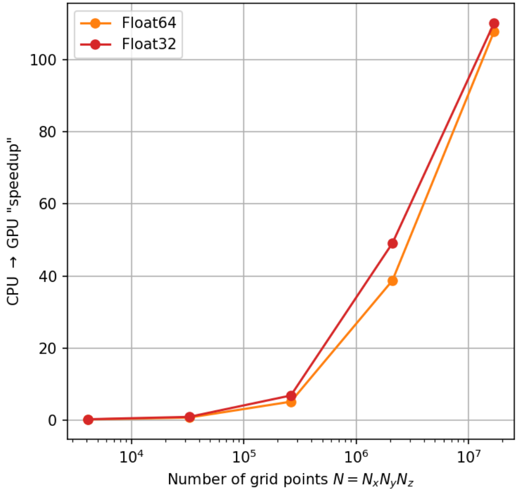
Simulation



```
@ Main c:\Users\dadoi\DataspellProjects\VS Code\Julia-Weather-Simulations\two_dimensional_diffusion.ipynb:7
Info: Iteration: 0800, time: 26.500, Δt: 4.12e-02, max(|u|) = 3.1e-01, wall time: 46.028 seconds
...
Info: Model iteration 1000 equals or exceeds stop iteration 1000.
@ Oceananigans.Simulations C:\Users\dadoi\.julia\packages\Oceananigans\OHYQj\src\Simulations\simulation.jl:18
Info: Iteration: 1000, time: 33.040, Δt: 4.02e-02, max(|u|) = 2.7e-01, wall time: 48.874 seconds
@ Main c:\Users\dadoi\DataspellProjects\VS Code\Julia-Weather-Simulations\two_dimensional_diffusion.ipynb:7
```

- Simulation

Performance



“Thank you for your attention.”

References

- Besard, Tim, Pieter Verstraete, and Bjorn De Sutter. "High-level GPU programming in Julia." arXiv preprint arXiv:1604.03410 (2016).
- Churavy, Valentin, et al. "Bridging HPC Communities through the Julia Programming Language." arXiv preprint arXiv:2211.02740 (2022).
- Hunold, Sascha, and Sebastian Steiner. "Benchmarking Julia’s Communication Performance: Is Julia HPC ready or Full HPC?" In 2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), pp. 20-25. IEEE, 2020.
- https://github.com/CliMA/Oceananigans.jl
- https://github.com/mak8427/Julia-Weather-Simulations