Native distributed and MPI parallelism in the high-level language Julia for quantum Monte Carlo

Mingrui Yang[1,2,3], Dr Elke Pahl[3,4,5] and Prof. Joachim Brand[1,2]

[1] New Zealand Institute for Advanced Study and Centre for Theoretical Chemistry and Physics, Massey University, Auckland, New Zealand
[2] Dodd-Walls Centre for Photonic and Quantum Technologies, New Zealand
[3] MacDiarmid Institute for Advanced Materials and Nanotechnology, New Zealand
[4] Department of Physics, University of Auckland, Auckland, New Zealand
[5] School of Natural and Computational Sciences and Centre for Theoretical Chemistry and Physics, Massey University, Auckland, New Zealand

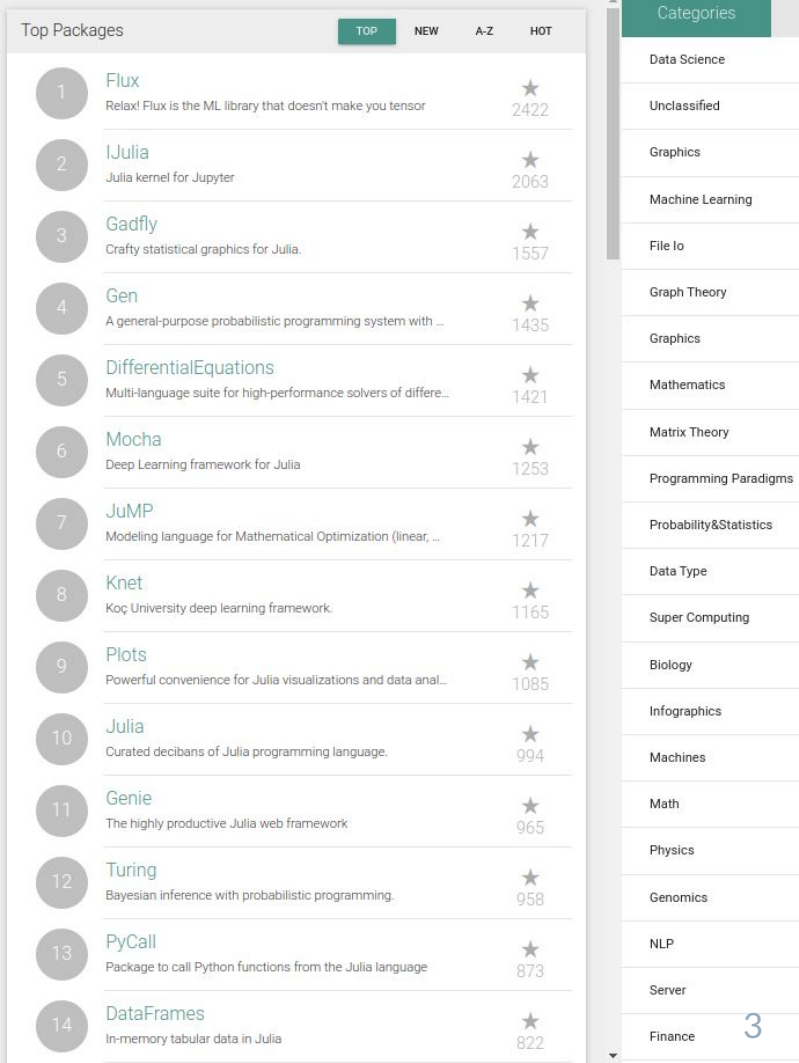
Outline

1. Julia on HPC
2. Paradigms of parallelism
3. Monte Carlo algorithm
4. Profiling the parallel Julia code on NeSI (NeSI consultancy project)

julia: the high level programming language

Key features:

- It's new: released in 2012; v1.0 in 2018
- High level language: fewer lines of code
- High performance: just-in-time (JIT) compiler
- 4225 (today) packages available in Julia =>
- Rapidly growing community
- Open source (it is free!)
- **Designed for parallel computing**



Parallel computing with julia

Options:

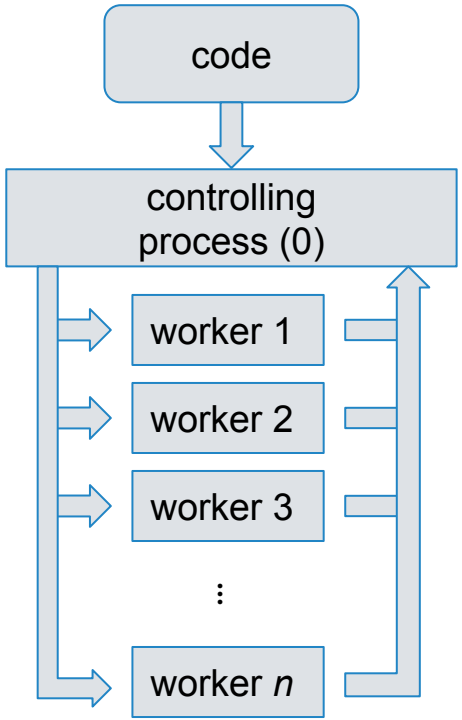
- **Distributed Computing (built-in with Distributed.jl, Julia standard library)**
  (distributed memory, works across nodes)

- Multi-Threading (built-in with @threads macro, available in Julia v1.3 or higher)
  (shared memory, works only within a single node)

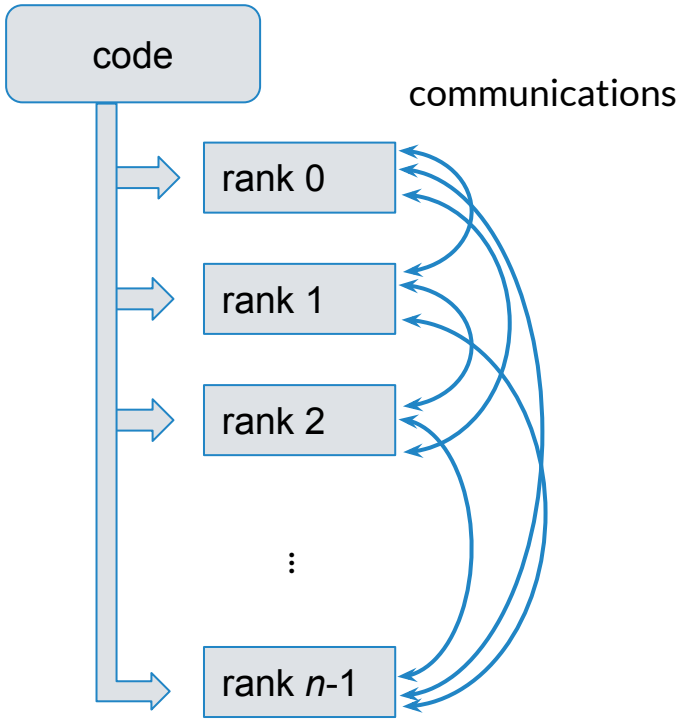
- **Message Passing Interface (with MPI.jl wrapper of external C MPI library)**
  (distributed memory, works across nodes with fast hardware communication)

Paradigms: Distributed vs MPI (for n processors)

Distributed.jl (built-in Julia package):



MPI.jl (with external MPI library):



Running Julia on HPC

- Julia is available as a module on NeSI's Mahuika & Maui, alternatively you can use the binary version

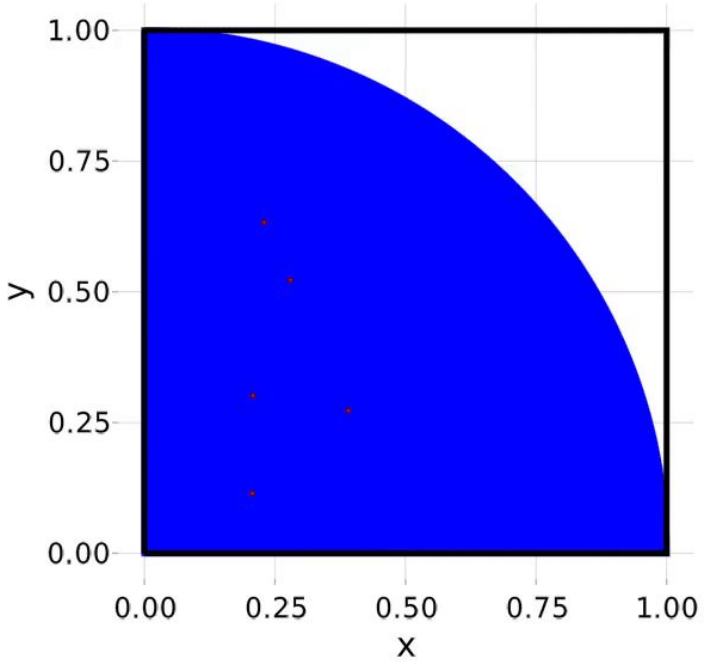
- For Distributed, use `--ntasks=1` and `--cpus-per-task=n` for slurm, and use `julia -p n <your_julia_code>.jl` to run your code

- For MPI, use `--ntasks=n` and `--cpus-per-task=1` in slurm, and use `srun julia <your_julia_code>.jl` to run your code

  A MPI module may also need to be loaded depends on the Julia build

Monte Carlo methods

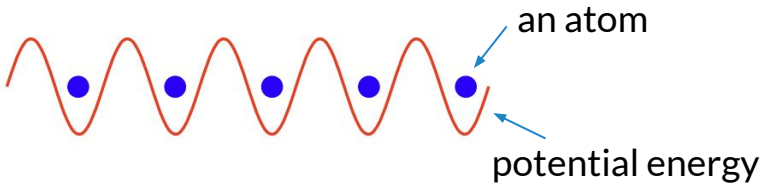
Estimating the value of π with random numbers:



$$\text{value of } \pi = 4\frac{(\text{area the } \blacktriangle)}{(\text{area the } \square)} \approx 4\frac{(\#\text{ of dots inside the } \blacktriangle)}{(\text{total } \#\text{ of dots})}$$

Monte Carlo sampling becomes very efficient for high-dimensional problems

Ultracold atom physics: experimentalists "put" particles into washboard-like potential:



The wave function:

$$\Psi = \sum_i c_i |D_i\rangle$$

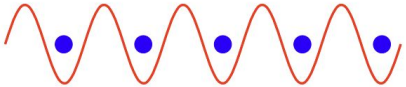
a coefficient

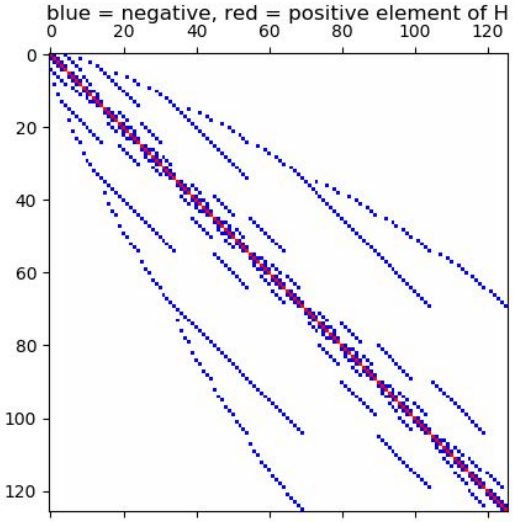
a configuration

Monte Carlo methods

Solve it as a (sparse) matrix eigenvalue problem

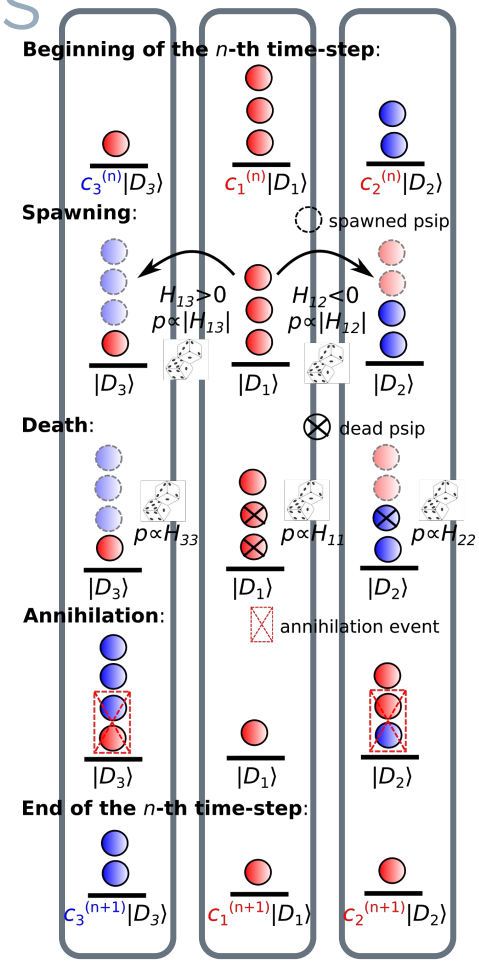
Schrödinger equation $\mathbf{H}\Psi = E\Psi$





Matrix dimension can get very large >> 10^{100} !!!

Full Configuration Interaction quantum Monte Carlo (FCIQMC) walker dynamics:



The wave function:

$$\Psi = \sum_i c_i |D_i\rangle$$

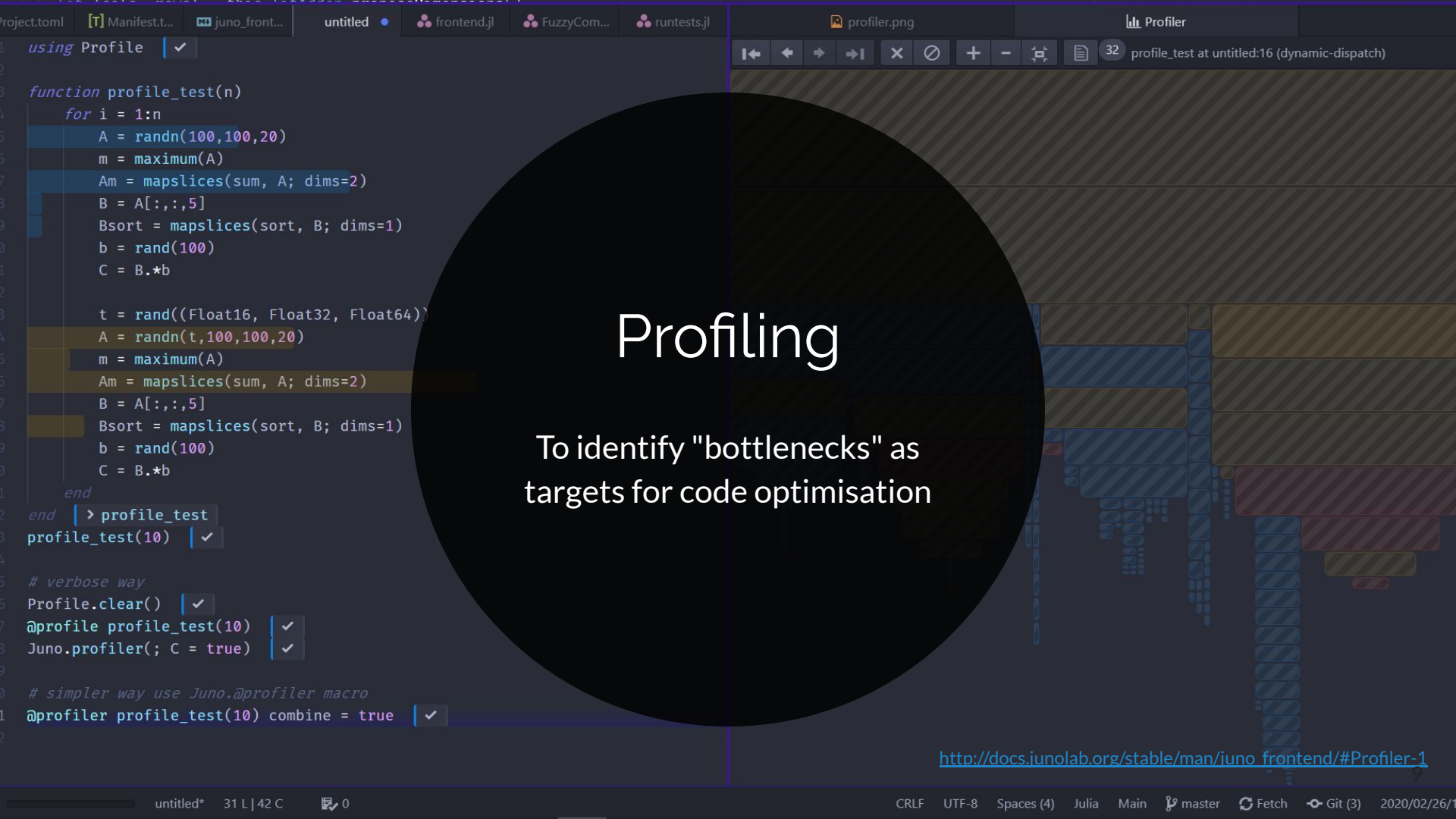
an integer

a bit "address" 01011011100...

Typically 10^6 to 10^9 walkers are used

Need parallelism and HPC!

$\|D_i\rangle$	: configuration
c_i	: coefficient
●	: (+ve) walker
●	: (-ve) walker

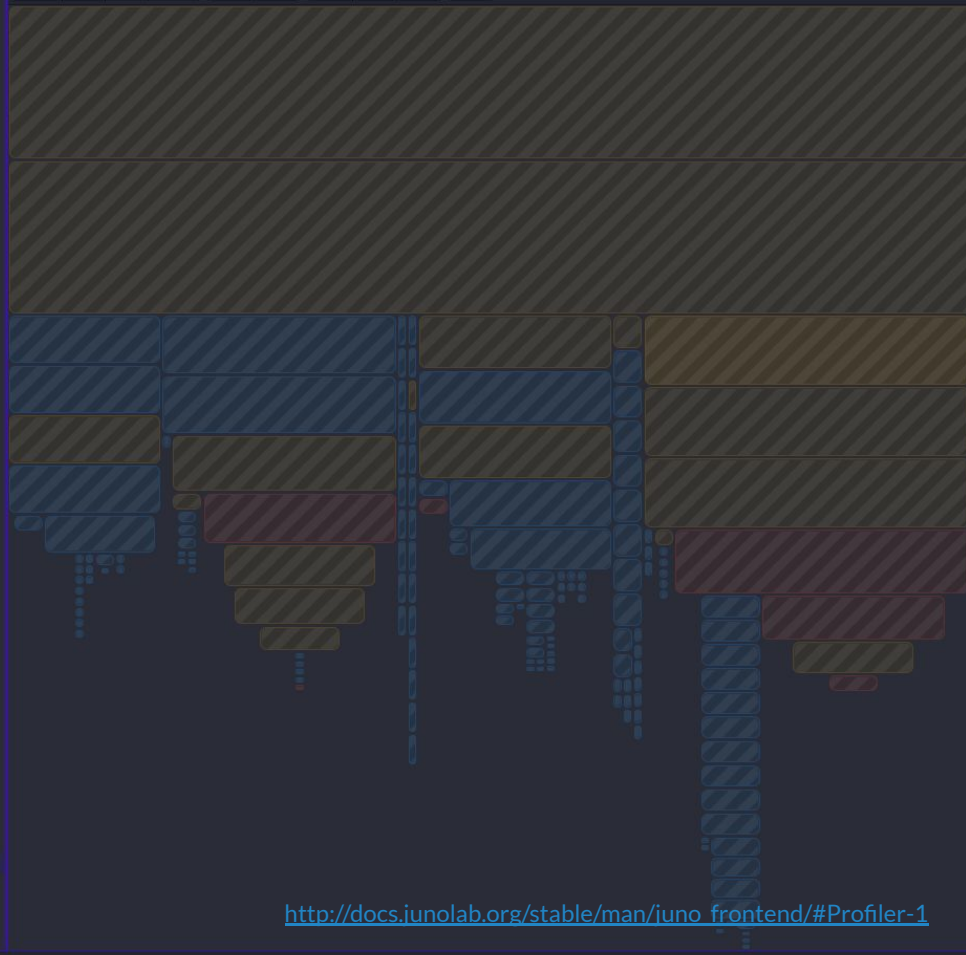

Project.toml
Manifest.t...
juno_front...
untitled
frontend.jl
FuzzyCom...
runtests.jl
using Profile
function profile_test(n)
for i = 1:n
A = randn(100,100,20)
m = maximum(A)
Am = mapslices(sum, A; dims=2)
B = A[:,:,5]
Bsort = mapslices(sort, B; dims=1)
b = rand(100)
C = B.*b
t = rand((Float16, Float32, Float64))
A = randn(t,100,100,20)
m = maximum(A)
Am = mapslices(sum, A; dims=2)
B = A[:,:,5]
Bsort = mapslices(sort, B; dims=1)
b = rand(100)
C = B.*b
end
end
profile_test
profile_test(10)
# verbose way
Profile.clear()
@profile profile_test(10)
Juno.profiler(; C = true)
# simpler way use Juno.@profiler macro
@profiler profile_test(10) combine = true
profiler.png
Profiler
32
profile_test at untitled:16 (dynamic-dispatch)
Profiling
To identify "bottlenecks" as targets for code optimisation
http://docs.junolab.org/stable/man/juno_frontend/#Profiler-1
untitled*
31 L | 42 C
0
CRLF
UTF-8
Spaces (4)
Julia
Main
master
Fetch
Git (3)
2020/02/26/1

Project.toml | Manifest.t... | juno_front... | untitled | frontend.jl | FuzzyCom... | runtests.jl

```
using Profile

function profile_test(n)
    for i = 1:n
        A = randn(100,100,20)
        m = maximum(A)
        Am = mapslices(sum, A; dims=2)
        B = A[:,:,5]
        Bsort = mapslices(sort, B; dims=1)
        b = rand(100)
        C = B.*b

        t = rand((Float16, Float32, Float64))
        A = randn(t,100,100,20)
        m = maximum(A)
        Am = mapslices(sum, A; dims=2)
        B = A[:,:,5]
        Bsort = mapslices(sort, B; dims=1)
        b = rand(100)
        C = B.*b
    end
end
profile_test(10)

# verbose way
Profile.clear()
@profile profile_test(10)
Juno.profiler(; C = true)

# simpler way use Juno.@profiler macro
@profiler profile_test(10) combine = true
```

profiler.png | Profiler

32 profile_test at untitled:16 (dynamic-dispatch)



http://docs.junolab.org/stable/man/juno_frontend/#Profiler-1

untitled* 31 L | 42 C 0 CRLF UTF-8 Spaces (4) Julia Main master Fetch Git (3) 2020/02/26/1

Profiling parallel Julia code (NeSI consultancy project)

With Chris Scott, Alexander Pletzer and Wolfgang Hayek

Challenges:

- Julia is new - limited guidance
- Some commercial tools do not support Julia yet
- Combined with parallelisation makes it harder for Julia native profiler
- Collect and visualise data on HPC (NeSI)

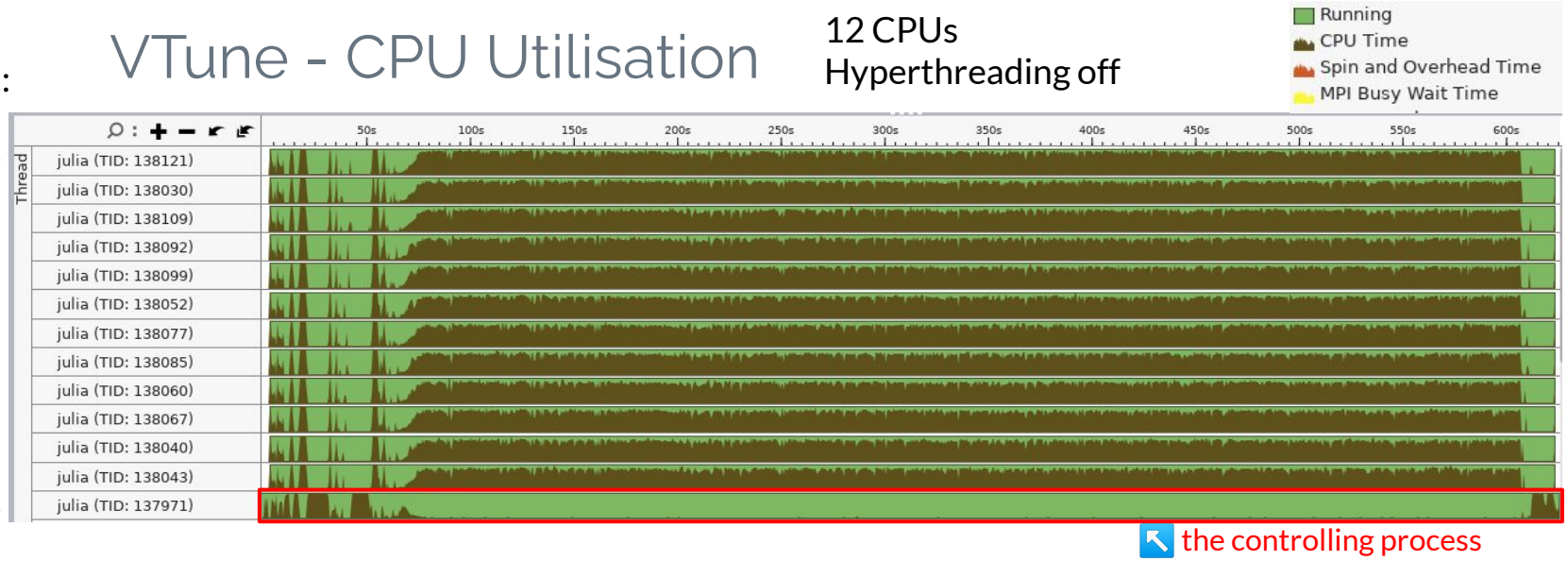
Outcomes:

- Slurm profiling - limited insight
- Intel® VTune™ Profiler works well for Julia with parallelism
- VTune's GUI is informative and fairly easy to navigate
- Need a special build for Julia to use VTune (won't work with binary version) - available now (v1.2; v1.4)

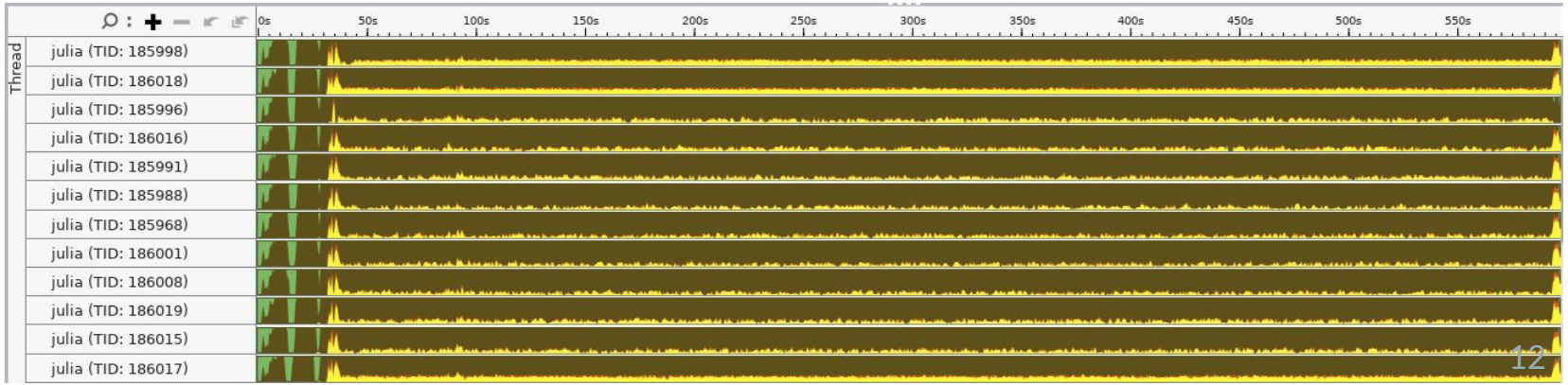
VTune - CPU Utilisation

Distributed.jl :

12 CPUs
Hyperthreading off



MPI.jl :



What's learned

Distributed:

- Small scale computation (small number of walkers)
- Single node (slower across nodes)
- Good for serial file IO
- Pure Julia code and no hardware dependency

MPI:

- Large scale (large number of walker with balanced load)
- Multiple nodes (fast InfiniBand interconnection)
- File IO in parallel (HDF5)
- Coding: more low-level thinking required

Profiling + NeSI consultancy:

- Optimised code (20x faster)
- *https://github.com/joachimbrand/Rimu.jl*
- *https://www.nesi.org.nz/services/consultancy*

Summary

- Julia is modern and powerful
- Multiple parallelisms available
- Many new things to explore
- NeSI consultancy is very helpful

Acknowledgement

Brand group and CTCP members at Massey

Chris Scott, Alexander Pletzer and Wolfgang Hayek

Thank you!