CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Rust programming language in high-performance computing

Rust Meetup

Michal Sudwoj

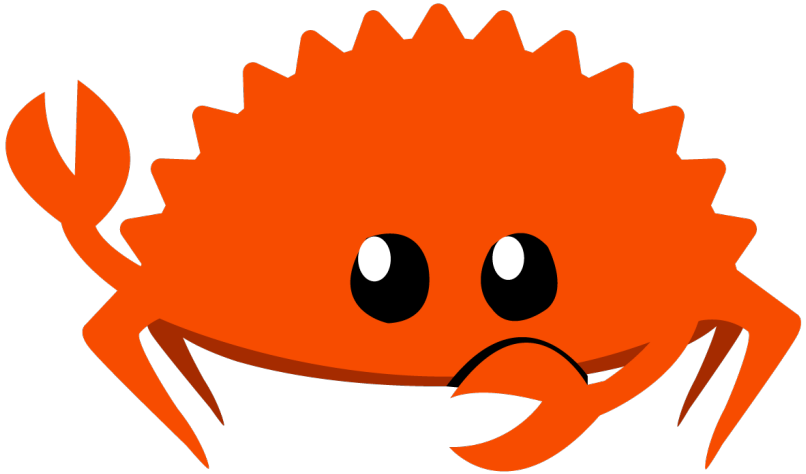
06.05.2022

Rust programming language in high-performance computing

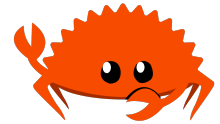
2022-05-06

Rust programming language in high-performance computing
Rust Meetup
Michal Sudwoj
06.05.2022

Before we begin…



Before we begin…



About me

- currently:
  - MSc Student in CSE at ETH Zürich
  - Data Engineer at Bank Vontobel
- before:
  - Intern at CSCS
  - Data Scientist at zkipster
  - BSc in CSE ar ETH Zürich

About me
currently:
MSc Student in CSE at ETH Zürich
Data Engineer at Bank Vontobel
before:
Intern at CSCS
Data Scientist at zkipster
BSc in CSE ar ETH Zürich

1. CSE: Computational Science and Engineering
2. from quantum chemistry to astrophysics
3. biology, meteorology, geology, …

What about you?

1. Software Engineers?
2. R&D?
3. PDE? Finite difference?

CSCS Intership: Part 1

In the first part, the Rust will be evaluated for potential usage at CSCS. In particular, the following questions should be tackled:

- how to **install and run Rust programs** with ”user access” rights on Piz Daint
- is MPI wrapper for Rust compatible with Cray’s implementation
- how to **interface Rust program with numerical libraries**, such as MKL, MAGMA, ScaLAPACK, cuBlasXt, etc.
- how to **write GPU-enabled application in Rust**; what are the complications or simplifications comparing to the C/C++/FORTRAN GPU applications
- how to debug and profile Rust-based programs
- how rich is the functionality of Rust, e.g. the availability of special mathematical functions, support of matrices or multi-dimensional arrays, etc.

CSCS Intership: Part 1

In the first part, the Rust will be evaluated for potential usage at CSCS. In particular, the following questions should be tackled:

- how to **install and run Rust programs** with ”user access” rights on Piz Daint
- is MPI wrapper for Rust compatible with Cray’s implementation
- how to **interface Rust program with numerical libraries**, such as MKL, MAGMA, ScaLAPACK, cuBlasXt, etc.
- how to **write GPU-enabled application in Rust**; what are the complications or simplifications comparing to the C/C++/FORTRAN GPU applications
- how to debug and profile Rust-based programs
- how rich is the functionality of Rust, e.g. the availability of special mathematical functions, support of matrices or multi-dimensional arrays, etc.

CSCS Intership: Part 2

In the second part, a **performance comparison between Rust and C/C++/FORTRAN** will be conducted, by idiomatically implementing a parallel distributed linear algebra algorithm or a scientific mini-app code in the target languages. The performance analysis is **not only limited to computational performance, but may include analysis of other factors, such as ease of implementation, number of bugs made, testability, readability, maintainability**, etc.

CSCS Intership: Part 2
In the second part, a **performance comparison between Rust and C/C++/FORTRAN** will be conducted, by idiomatically implementing a parallel distributed linear algebra algorithm or a scientific mini-app code in the target languages. The performance analysis is **not only limited to computational performance, but may include analysis of other factors, such as ease of implementation, number of bugs made, testability, readability, maintainability**, etc.

Part 1: Installation

```
> curl https://sh.rustup.rs -sSf | sh
> rustup toolchain install nightly
> rustup target add nvptx64-nvidia-cuda
> # On Piz Daint
> export
>   CARGO_TARGET_X86_64_UNKNOWN_LINUX_GNU_RUSTFLAGS="
>     -C target-cpu=haswell
>     -C relocation-model=dynamic-no-pic
>   "
>   CARGO_TARGET_NVPTX64_NVIDIA_CUDA_RUSTFLAGS="
>     -C target-cpu=sm_60
>     -C target-feature=+sm_60,+ptx60
>     -C relocation-model=dynamic-no-pic
>   "
>   MPICC=cc
> cargo install ptx-linker
```

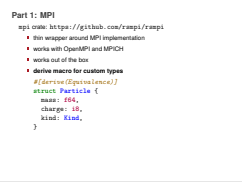

Part 1: Installation
> curl https://sh.rustup.rs -sSf | sh
> rustup toolchain install nightly
> rustup target add nvptx64-nvidia-cuda
> # On Piz Daint
> export
> CARGO_TARGET_X86_64_UNKNOWN_LINUX_GNU_RUSTFLAGS="
> -C target-cpu=haswell
> -C relocation-model=dynamic-no-pic
> "
> CARGO_TARGET_NVPTX64_NVIDIA_CUDA_RUSTFLAGS="
> -C target-cpu=sm_60
> -C target-feature=+sm_60,+ptx60
> -C relocation-model=dynamic-no-pic
> "
> MPICC=cc
> cargo install ptx-linker

Part 1: MPI

`mpi` crate: `https://github.com/rsmpi/rsmpi`

- thin wrapper around MPI implementation
- works with OpenMPI and MPICH
- works out of the box
- **derive macro for custom types**

```
#[derive(Equivalence)]
struct Particle {
  mass: f64,
  charge: i8,
  kind: Kind,
}
```

Part 1: Interfacing with C

- `bindgen` crate
- existing bindings to BLAS, LAPACK, cublas, MPI, …

Part 1: Interfacing with C
- bindgen crate
- existing bindings to BLAS, LAPACK, cublas, MPI, ...

Part 2: Toy Problem

- fourth-order numerical diffusion in xy-plane

$$\frac{\partial \phi}{\partial t} = -\alpha_4 \nabla^4_{xy} \phi = \underbrace{-\alpha_4 \Delta^2_{xy} \phi}_{\texttt{inline}} = \underbrace{-\alpha_4 \Delta_{xy} (\Delta_{xy} \phi)}_{\texttt{laplap}}$$

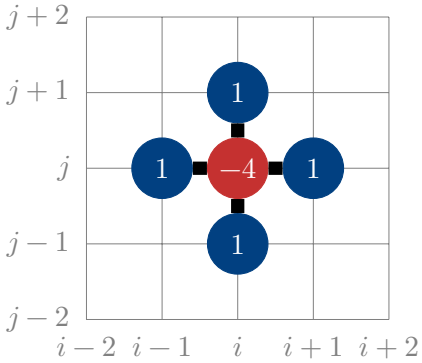
- on unit cube
- boundary conditions: $\partial_\Omega = 0$

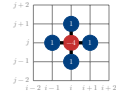
Part 2: Toy Problem
- fourth-order numerical diffusion in xy-plane
$\frac{\partial \phi}{\partial t} = -\alpha_4 \nabla^4_{xy} \phi = \underbrace{-\alpha_4 \Delta^2_{xy} \phi}_{\texttt{inline}} = \underbrace{-\alpha_4 \Delta_{xy} (\Delta_{xy} \phi)}_{\texttt{laplap}}$
- on unit cube
- boundary conditions: $\partial_\Omega = 0$

1. Xue (2000)
2. Used in weather simulations as a smoothing kernel
3. Reference implementation available - Oliver Fuhrer

$$\phi^{n+1}_{i,j,k} = \phi^{n}_{i,j,k} - \frac{\alpha_4 \Delta t}{\Delta x \Delta y} \Delta \Big(-4\phi^{n}_{i,j,k} + \phi^{n}_{i-1,j,k} + \phi^{n}_{i+1,j,k} + \phi^{n}_{i,j-1,k} + \phi^{n}_{i,j+1,k} \Big)$$

laplap





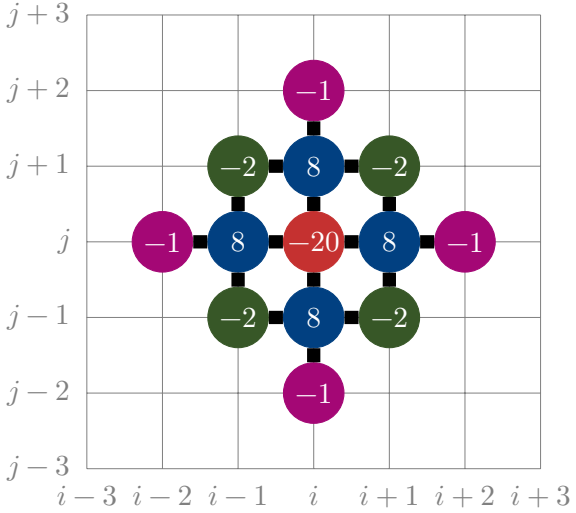
inline

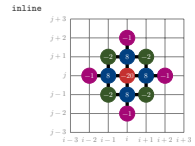
$$\phi^{n+1}_{i,j,k} = \phi^n_{i,j,k} - \frac{\alpha_4 \Delta t}{(\Delta x)^2 (\Delta y)^2} \Big(\\ - 20\phi^n_{i,j,k} \\ + 8\phi^n_{i-1,j,k} + 8\phi^n_{i+1,j,k} + 8\phi^n_{i,j-1,k} + 8\phi^n_{i,j+1,k} \\ - 2\phi^n_{i-1,j-1,k} - 2\phi^n_{i-1,j+1,k} - 2\phi^n_{i+1,j-1,k} - 2\phi^n_{i+1,j+1,k} \\ - \phi^n_{i-2,j,k} - \phi^n_{i+2,j,k} - \phi^n_{i,j-2,k} - \phi^n_{i,j+2,k} \Big)$$

inline

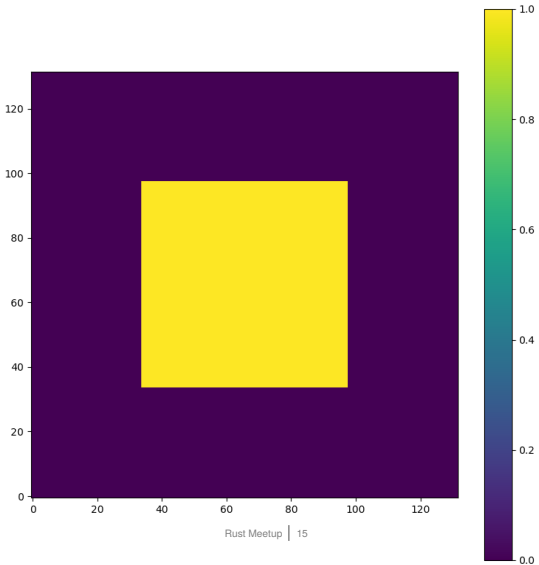
1. What do you think - which version is faster?

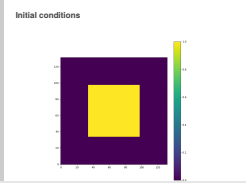
inline



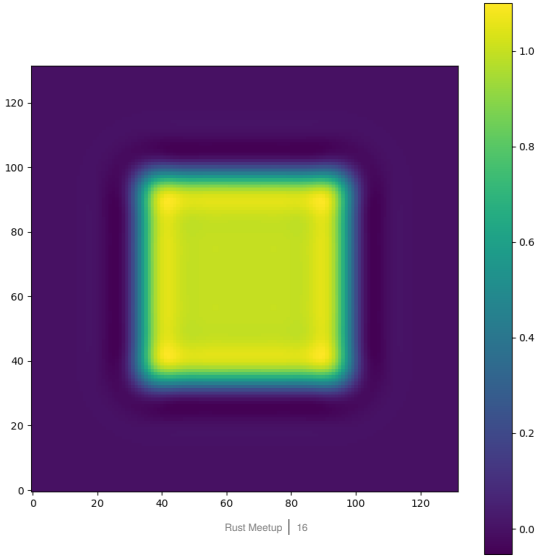


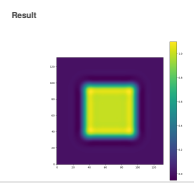
Initial conditions





Result





Measurement setup

- Fortran, C++, Rust
- Algorithms: `laplap`, `inline`
- Toolchains
  - Fortran, C++: GNU, Cray, Intel, PGI
  - Rust: `rustc`
- Sequential
- Parallel
  - Fortran, C++: OpenMP
  - Rust: Rayon
- GPU
  - Fortran, C++: OpenACC, OpenMP offloading, CUDA
  - Rust: Accel

Measurement setup
- Fortran, C++, Rust
- Algorithms: laplap, inline
- Toolchains
  - Fortran, C++: GNU, Cray, Intel, PGI
  - Rust: rustc
- Sequential
- Parallel
  - Fortran, C++: OpenMP
  - Rust: Rayon
- GPU
  - Fortran, C++: OpenACC, OpenMP offloading, CUDA
  - Rust: Accel

1. MPI works, but not enough time to debug the partitioner

Measurement setup

$\Longrightarrow$ 84 language-compiler-algorithm combinations

32 versions did not compile or run :(

$\Longrightarrow$ 52 versions $\times$ 4 grid sizes

$\Longrightarrow$ 524 measurements (1, 2, 4, 8, 12 CPU cores; GPU)

Measurement setup
⟹ 84 language-compiler-algorithm combinations
32 versions did not compile or run :(
⟹ 52 versions × 4 grid sizes
⟹ 524 measurements (1, 2, 4, 8, 12 CPU cores; GPU)

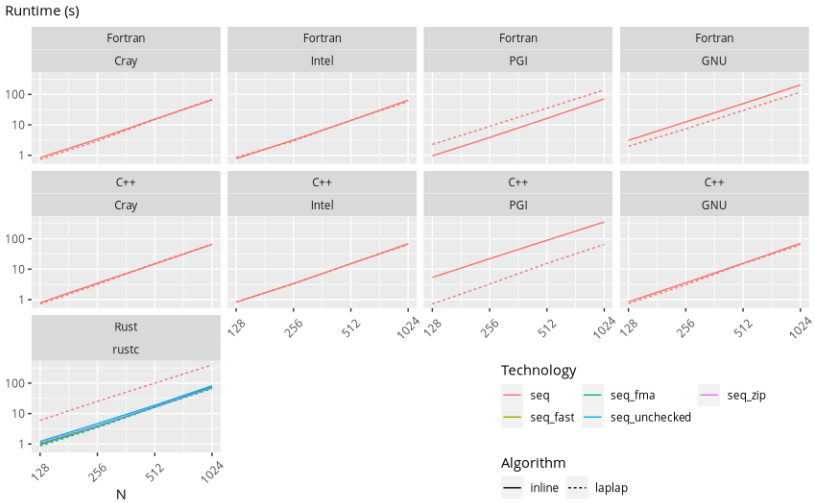
- arrays in column-major order
- −O3 or equivalent
- no LTO
- optimization reports
- shared libraries
- C interface

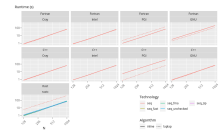
```
void diffuse(
  float * in_field,
  float * out_field,
  size_t  nx,
  size_t  ny,
  size_t  nz,
  size_t  num_halo,
  float   alpha,
  size_t  num_iter
)
```

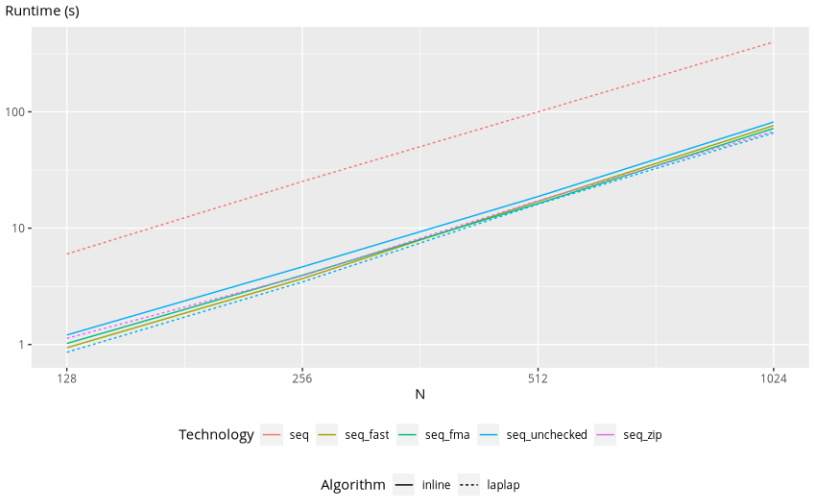

- arrays in column-major order
- −O3 or equivalent
- no LTO
- optimization reports
- shared libraries
- C interface

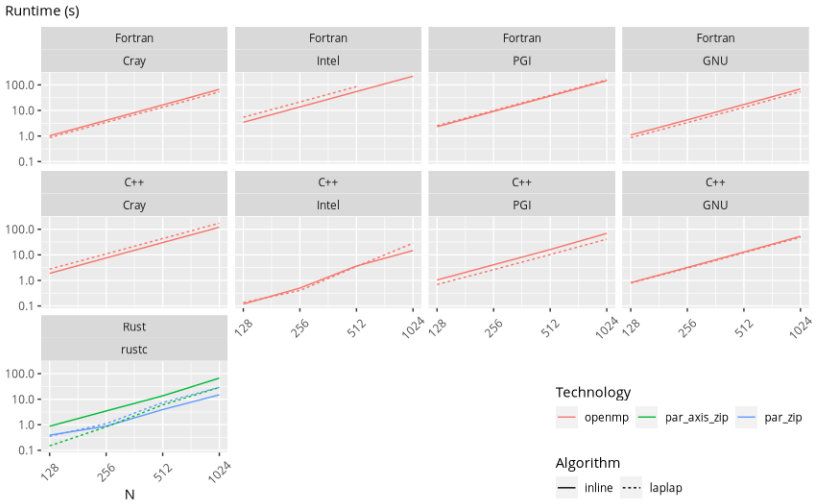
```
void diffuse(
  float * in_field,
  float * out_field,
  size_t  nx,
  size_t  ny,
  size_t  nz,
  size_t  num_halo,
  float   alpha,
  size_t  num_iter
)
```

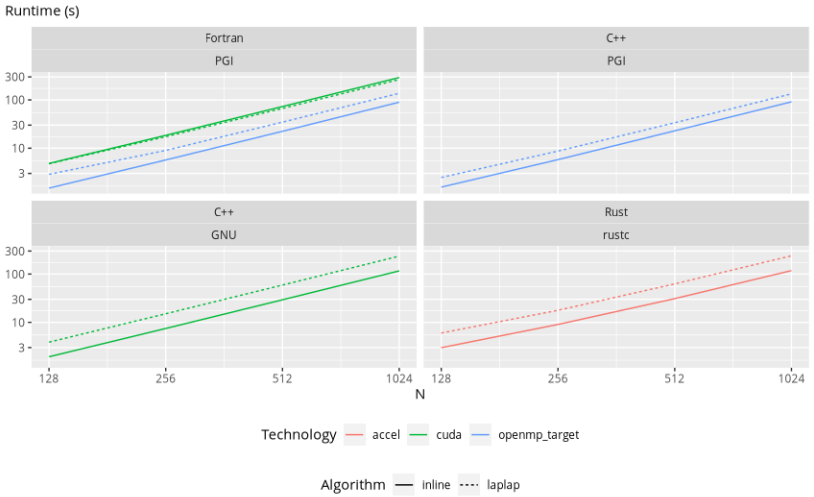
1. Show code on gitlab

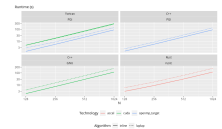

Runtime (s)
Fortran
Cray
Intel
PGI
GNU
C++
Rust
rustc
100
10
1
128
256
512
1024
N
Technology
seq
seq_fma
seq_zip
seq_fast
seq_unchecked
Algorithm
inline
laplap




Runtime (s)
100
10
1
128
256
512
1024
N
Technology
seq
seq_fast
seq_fma
seq_unchecked
seq_zip
Algorithm
inline
laplap


Runtime (s)
Fortran
Cray
Fortran
Intel
Fortran
PGI
Fortran
GNU
C++
Cray
C++
Intel
C++
PGI
C++
GNU
Rust
rustc
100.0
10.0
1.0
0.1
128
256
512
1024
N
Technology
openmp
par_axis_zip
par_zip
Algorithm
inline
laplap


Runtime (s)
Fortran
PGI
C++
PGI
C++
GNU
Rust
rustc
300
100
30
10
3
128
256
512
1024
N
Technology
accel
cuda
openmp_target
Algorithm
inline
laplap



Conclusions

- \+ Rust fast enough for scientific software
- \+ Safer code
- \+ Clearer error messages
- − Support for certain features is lacking
- ∼ All languages sometimes lead to lots of boilerplate

Conclusions
+ Rust fast enough for scientific software
+ Safer code
+ Clearer error messages
− Support for certain features is lacking
∼ All languages sometimes lead to lots of boilerplate

1. Rust compiler does good job of vectorizing code
2. Rust: clear what you get
3. Fortran/C++: different support, many bugs
4. GNU doesn't warn about lack of offloading
5. Cray sometimes silently generates invalid PTX code

Recommendations

- in 2020
  - Rust for frontend/driver
- hopes for after 2020
  - GPU support
  - Evolve `ndarray`
  - ScaLAPACK bindings, …
- $\Longrightarrow$ continue to pursue Rust in HPC
  - it has potential

See:

- `https://www.arewelearningyet.com/scientific-computing/`
- `https://www.arewelearningyet.com/gpu-computing/`

Recommendations
- in 2020
  - Rust for frontend/driver
- hopes for after 2020
  - GPU support
  - Evolve ndarray
  - ScaLAPACK bindings, …
- ⟹ continue to pursue Rust in HPC
  - it has potential

See:
- https://www.arewelearningyet.com/scientific-computing/
- https://www.arewelearningyet.com/gpu-computing/

1. With Cray and Intel moving to LLVM, cross-language LTO soon?

Two years later …

not much has changed :(

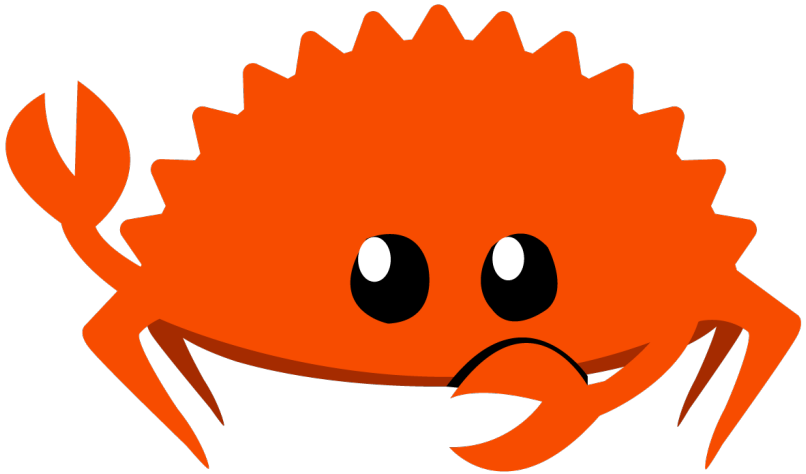
not much has changed :(

Two years later …

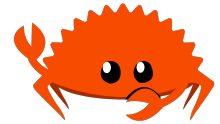
- `cargo-cmake` → `corrosion`
- `rust-cuda` → `rust-gpu`
- `nvptx64-nvidia-cuda` still a Tier 2 target

Two years later …
- cargo-cmake → corrosion
- rust-cuda → rust-gpu
- nvptx64-nvidia-cuda still a Tier 2 target

Questions?



Questions?

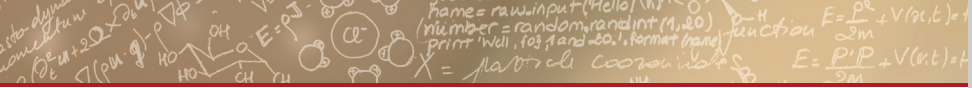


Keep in touch?

- michal@sudwoj.name
- https://github.com/westernmagic/rust-in-hpc
- www.linkedin.com/in/michalsudwoj

Keep in touch?

Keep in touch?
- michal@sudwoj.name
- https://github.com/westernmagic/rust-in-hpc
- www.linkedin.com/in/michalsudwoj



Thank you for you attention

Thank you for you attention