Karlsruher Institut für Technologie

SCC

# Incremental MPI Parallelization of a Julia Functional Renormalization Group code: a case study

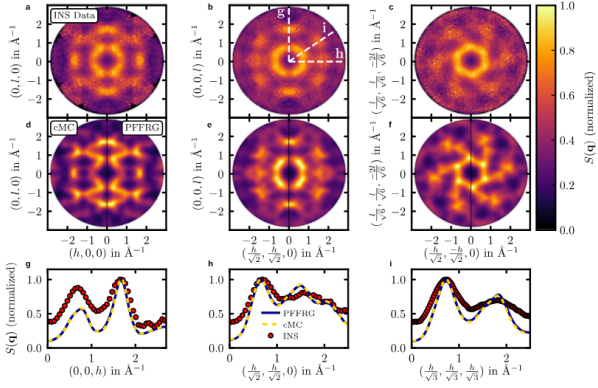
Michele Mesiti | Feb 26, 2025

# Quick facts about PMFRG.jl

- Written by Nils Niggemann while PhD at FU Berlin, now Postdoc at ICTP (Trieste)
- Paper on the method: “Frustrated quantum spins at finite temperature: Pseudo-Majorana functional renormalization group approach (LINK)”
- Language: Julia
- HPC techniques used (originally): Task-Based Multithreading, Vectorization

# Application domain: Spin Structure Factors

Dynamics of $K_2Ni_2(SO_4)_3$ governed by proximity to a 3D spin liquid model




(INS: Inelastic Neutron Scattering)

# Human Context

- HiRSE project (then, still HiRSE_PS)
- Some very mature codebases, some in its infancy -> PFFRG
- MPI parallelization with mutual benefit:
  - Interest from the code owners in MPI parallelization - MPI.jl was quite clearly the way to go for them:
    - previous C++ codes using OpenMP + MPI
    - MPI.jl being regarded as more performant than alternatives in the Julia ecosystem
  - Interest from M.M. in trying Julia and get experience in it

# Application domain and computational approach

## Application Domain

Heisenberg models ($\vec{S}_i$ = electron spin):

$$H = \sum_{ij} J_{ij} \vec{S}_i \vec{S}_j \tag{1}$$

$J_{i,j}$: interaction between spins in different lattice sites
No analytical & direct methods to compute the correlation function $\Gamma$ exist but we have approximate and "indirect" methods.

## Computational approach: Functional Renormalization Group in a nutshell

- We introduce a parameter $\Lambda$ which is inversely related to the maximum distance at which we allow the system to have its own dynamics (e.g., interact)
- We know the values of $\Gamma$ when $\Lambda \to \infty$ (no long-range interaction)
- We know the $\frac{d\Gamma(\Lambda)}{d\Lambda}$
- We try to compute the real value of $\Gamma = \Gamma(0)$ by integrating the corresponding ODE

# Computational challenges

## Data structures

- $\frac{d\Gamma(\Lambda)}{d\Lambda}$ is easily computed from 4D arrays named $X_{a,b,c}$ and $\tilde{X}_{a,b,c,d}$, with indices $(Rij, is, it, iu)$
  - $Rij < \sim 100$
  - $is,it,iu < N $ N \sim 50$

  See Eqs. 50 of the paper.

## "Extreme pareto"

98% of the computational load is in 2% of the code

- The computation of the derivative is THE numerical challenge ($ FLOP \propto N^4 $) and scales well (*) with the number of processors and the number of nodes

# Computational challenge

```
for Rij = 1:Npairs
    #Perform summation on this temp variable before writing to State array as Base.setindex! proved to be a bottleneck!
    Xa_sum = 0.0
    Xb_sum = 0.0
    Xc_sum = 0.0
    @turbo unroll = 1 for k_spl = 1:Nsum[Rij]
        #loop over all Nsum summation elements defined in geometry. This inner loop is responsible for most of the computational effort!
        ki = S_ki[k_spl, Rij]
        kj = S_kj[k_spl, Rij]
        m  = S_m [k_spl, Rij]
        xk = S_xk[k_spl, Rij]

        Ptm = Props[xk, xk] * m
        Xa_sum += (+Va12[ki] * Va34[kj] + Vb12[ki] * Vb34[kj] * 2) * Ptm
        Xb_sum +=
            (+Va12[ki] * Vb34[kj] + Vb12[ki] * Va34[kj] + Vb12[ki] * Vb34[kj]) * Ptm
        Xc_sum += (+Vc12[ki] * Vc34[kj] + Vc21[ki] * Vc43[kj]) * Ptm
    end
    X.a[Rij, is, it, iu] += Xa_sum
    X.b[Rij, is, it, iu] += Xb_sum
    X.c[Rij, is, it, iu] += Xc_sum
end
```

# Do you really want MPI?

Ideally:

- Before going on multiple nodes, you should make sure you use one well
- Before going on multiple cores, you should make sure you use one well

Not just for your Core*Hour budget, but for the Planet.
From the HLRS Node-Level Performance Engineering course page:

> *Even application developers who are fluent in OpenMP and MPI often lack a good grasp of how much performance could at best be achieved by their code. This is because parallelism takes us only half the way to good performance. Even worse, slow serial code tends to scale very well, hiding the fact that resources are wasted.*

# Do you really want MPI?

- Performance optimization data missing,
- but evidently some work had been done on optimization:
  - unintuitive nested loop order
  - creation of buffers to avoid `setindex` and `getindex` bottlenecks (according to code comments)

# Learning Performance Characterization in Julia the Hard Way

Weeks of Misery spent on:

- Chasing sources of lack of reproducibility in benchmark results:
  - not following the Manifest, perhaps? (AKA not knowing what your code actually depends upon) Should the Manifest be under version control?
  - thread pinning, perhaps? (Thanks, ThreadPinning.jl)
  - performance governors, perhaps?(Thanks, likwid-setFrequencies )
- Trying to collect performance counters with LIKWID.jl (Serializing the results and trying to read them again fails - possibly wrong approach, or maybe use JLD2?)
- Profiling with pprof.jl would produce malformed `profile.pb.gz` (solution found many months later)

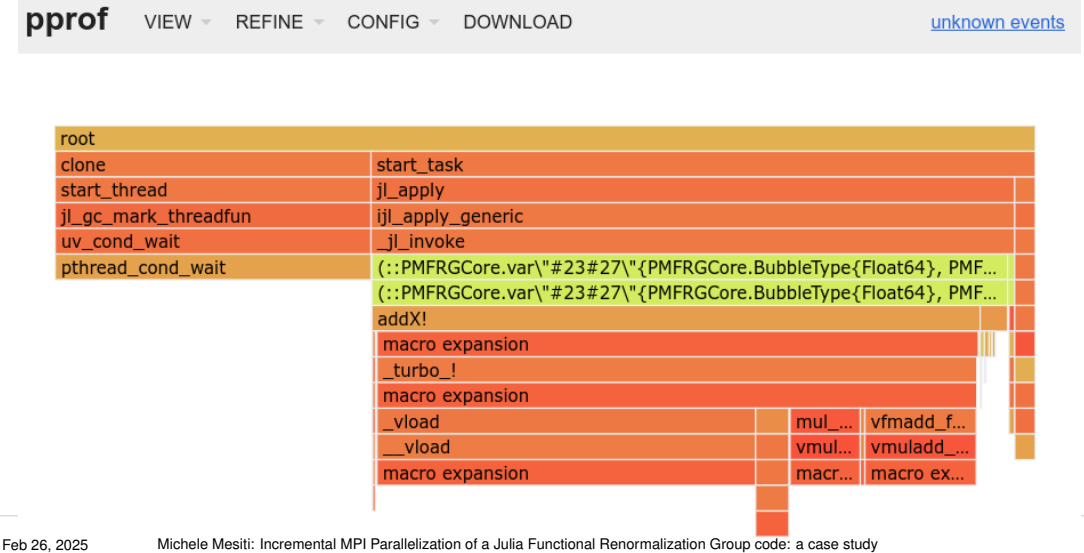
Settled on TimerOutputs.jl

# Learning Performance Characterization in Julia the Hard Way: Downfall + Tower of Babel Edition

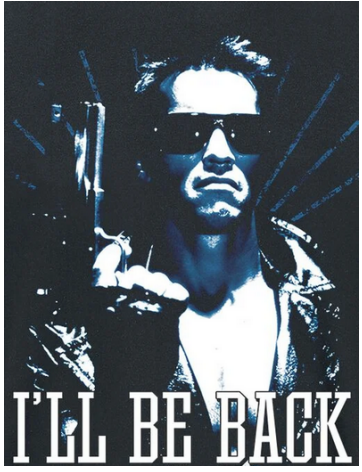
Karlsruher Institut für Technologie

1. Wednesday, week 1: Reference benchmark runs in ~14s, in-node scaling unsatisfying
2. Friday, week 1: Reference benchmark runs in ~20s, in-node scaling better
3. week 2: Heard that the Downfall mitigation might have affected scatter/gather operations *a little*. For the lack of better ideas, I ask for information
4. week 3: Sysadmins revert microcode change on one of the 2 nodes on the test cluster. We miscommunicate, and I get the information that they reverted on the other node
5. I go crazy for a week:
   - discover that changes made with `likwid-setFrequencies` are not reset after job terminates.
   - sysadmins take `likwid-setFrequencies` out of reach of ordinary users
6. A colleague shows me the `microcode` field in `/proc/cpuinfo` and the misunderstanding happened at 3. is solved
7. I ask for help in the discourse

# Attempt at performance characterization and in-node scaling



# Attempt at performance characterization and in-node scaling



# MPI Implementation Constraints

- MPI should be optional
- MPI code should be confined away from where domain scientists (non MPI-savvy) would see it

Decided to use the (then) new “Package Extension” feature.

# MPI Implementation - Preparation

I need to do some characterization tests! And of course I wrote my tool for that:

## Recorder.jl

From the Readme of Recorder.jl:

```
using Recorder
using MyModule

function deep_in_the_callstack_in_nested_loops_and_without_tests()
    [...]
    res = @record func(a,b,c)
    [...]
end

Take out a lot of boilerplate to generate this kind of test cases.
```

(I wanted to do something like this since long).

# MPI Implementation

- Each MPI rank computes the result for a section of the X arrays sub-range in the indices *is*, *it* and *iu* (ranges in *iu* are NOT equal in size for different ranks -> Load balancing requires care)
- Data is communicated in an AllToAll fashion

```
for root = 0:(nranks-1)
    isrange, itrange, iurange_restrict = all_ranges[root+1]
    iurange_abc = Par.Options.usesymmetry ? iurange_restrict : iurange_full

    MPI.Bcast!((@view X.a[:, isrange, itrange, iurange_abc]), root, MPI.COMM_WORLD)
```

# Results - 16 nodes - DP5

| | | Time | | | Allocations | | |
|---|---|---|---|---|---|---|---|
| Tot / % measured: | | 999s / 99.5% | | | 120GiB / 98.5% | | |
| Section | ncalls | time | %tot | avg | alloc | %tot | avg |
| total solver | 1 | 994s | 100.0% | 994s | 118GiB | 100.0% | 118GiB |
| getDeriv! | 409 | 970s | 97.5% | 2.37s | 114GiB | 96.9% | 287MiB |
| getXBubble! | 409 | 882s | 88.7% | 2.16s | 97.5MiB | 0.1% | 244KiB |
| partition | 409 | 578s | 58.2% | 1.41s | 46.2MiB | 0.0% | 116KiB |
| communication | 409 | 303s | 30.5% | 741ms | 19.6MiB | 0.0% | 49.0KiB |
| get_ranges | 409 | 92.4ms | 0.0% | 226us | 31.4MiB | 0.0% | 78.7KiB |
| rebuildStateStruct! | 409 | 46.4s | 4.7% | 113ms | 114GiB | 96.7% | 286MiB |
| repackStateVector! | 409 | 20.5s | 2.1% | 50.0ms | 658KiB | 0.0% | 1.61KiB |
| addToVertexFromBubble! | 409 | 8.86s | 0.9% | 21.7ms | 17.8MiB | 0.0% | 44.6KiB |
| symmetrizeBubble! | 409 | 3.44s | 0.3% | 8.41ms | 35.6MiB | 0.0% | 89.2KiB |
| get_Self_Energy! | 409 | 508ms | 0.1% | 1.24ms | 49.1MiB | 0.0% | 123KiB |
| symmetrizeVertex! | 409 | 212ms | 0.0% | 519us | 22.6MiB | 0.0% | 56.5KiB |
| getDFint! | 409 | 2.18ms | 0.0% | 5.33us | 0.00B | 0.0% | 0.00B |
| workspace | 409 | 133us | 0.0% | 325ns | 0.00B | 0.0% | 0.00B |
| setup | 409 | 49.3us | 0.0% | 120ns | 0.00B | 0.0% | 0.00B |

# Results - 4 Nodes - VCABM

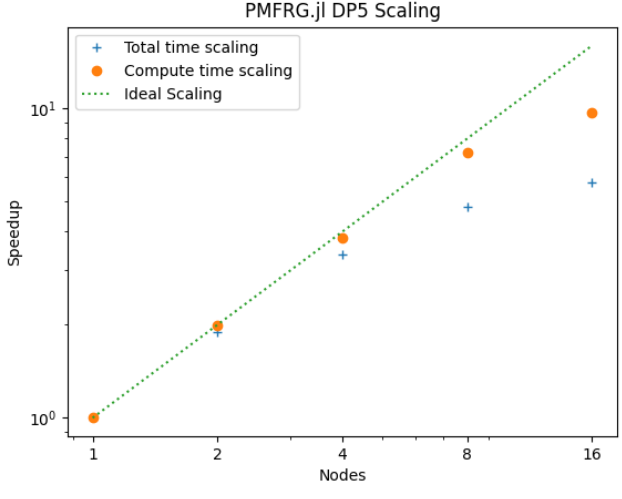
| | | Time | | | Allocations | | |
|---|---|---|---|---|---|---|---|
| Tot / % measured: | | 1655s / 99.9% | | | 100GiB / 98.2% | | |
| Section | ncalls | time | %tot | avg | alloc | %tot | avg |
| total solver | 1 | 1653s | 100.0% | 1653s | 97.7GiB | 100.0% | 97.7GiB |
| getDeriv! | 278 | 1083s | 65.5% | 3.90s | 77.9GiB | 79.7% | 287MiB |
| getXBubble! | 278 | 1026s | 62.1% | 3.69s | 145MiB | 0.1% | 535KiB |
| partition | 278 | 952s | 57.6% | 3.42s | 139MiB | 0.1% | 512KiB |
| communication | 278 | 67.3s | 4.1% | 242ms | 3.33MiB | 0.0% | 12.2KiB |
| get_ranges | 278 | 11.7ms | 0.0% | 42.2us | 2.64MiB | 0.0% | 9.72KiB |
| rebuildStateStruct! | 278 | 29.8s | 1.8% | 107ms | 77.7GiB | 79.5% | 286MiB |
| repackStateVector! | 278 | 12.7s | 0.8% | 45.6ms | 447KiB | 0.0% | 1.61KiB |
| addToVertexFromBubble! | 278 | 5.96s | 0.4% | 21.5ms | 12.1MiB | 0.0% | 44.6KiB |
| symmetrizeBubble! | 278 | 2.27s | 0.1% | 8.17ms | 24.2MiB | 0.0% | 89.2KiB |
| get_Self_Energy! | 278 | 349ms | 0.0% | 1.25ms | 33.4MiB | 0.0% | 123KiB |
| symmetrizeVertex! | 278 | 159ms | 0.0% | 572us | 15.3MiB | 0.0% | 56.5KiB |
| getDFint! | 278 | 1.55ms | 0.0% | 5.59us | 0.00B | 0.0% | 0.00B |
| workspace | 278 | 203us | 0.0% | 731ns | 0.00B | 0.0% | 0.00B |
| setup | 278 | 30.1us | 0.0% | 108ns | 0.00B | 0.0% | 0.00B |

# Trying to fix scaling with VCABM

- made pull request to OrdinaryDiffEq to enable multithreading in all VCABM methods
- Use PencilArrays.jl to make sure that the “internal solver” computation can be split between ranks

NOTHING changed.

# Scaling Plot - DP5



# Funny Issues along the road

Problems I experienced beforehand:

- Autoformatting of expressions like $-x$ would fail
- OpenMPI, UCX and the Julia Garbage collectors conjure to make your program segfault (known problem but I misread the docs or got tricked anyway)
- Loading HDF5 afer MPI.Init() would cause linker problems, but the opposite would not.

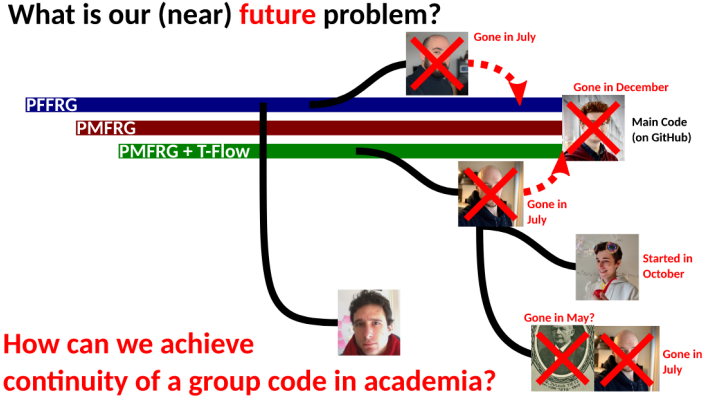
Problems that other people had, they solved, and I used ther solution:

- Pencil arrays did not originally implement global reductions across MPI ranks
- (possibly many others)

**What have I learned:**

*Packages only provide "leaky" abstractions, using a package does not mean you can ignore its internals*

# Human Context, revisited




(From Matias' Gonzalez HiRSE talk, 26 Nov 2024, Berlin)

# Recap and Outlook

Work done:

- Some performance characterization (`@#$%@#$^!!#@!!`)
- Characterization tests (with Recorder.jl)
- MPI Parallelization (Derivative computation)
- Refactoring to streamline development
- MPI Parallelization of Solver (?) with PencilArrays.jl
  - Good scaling with DP5
  - Problems with VCABM

To Do:

- Clean and publish the code (probably in SciPost Physics Codebases)
- Try to merge al lines of work together

Paused for now:

- Performance Optimization

## Recap and Outlook

Thank you for listening.

# Computational approach: possible improvements I

## Symmetries in the data structures

X has symmetries, which are used to reduce the computational load, but make efficient load balancing a challenge (at least, in my implementation). In particular:

- for $is + it + iu = 2n$ (or $2n + 1$), $X[Rij, is, it, iu] = 0$, but data is communicated anyway
- for $X_{a,b,c}$, $X[\ldots, \ldots, it, iu] = X[\ldots, \ldots, iu, it]$: This makes load balancing difficult, at least in the current implementation

## How to exploit the symmetries?

- Use 1D arrays instead of 4D, assign contiguous segments to MPI ranks, obtaining better load balance
- write custom getter/setter functions so that the single index is mapped to $(Rij, is, it, iu)$
  - either define the mapping function as standalone
  - or create getter as a method of *getindex* and setter as a method of *getindex*! (see the Julia Array Interface) - More elegant but I am not sure this can be done, or if it is a good idea.

# Computational approach: possible improvements II

## Optimize SpinFRGLattices.jl

SpinFRGLattices produces the lookup/connectivity tables, it might be the key to improve the performance (In a sense, the “Extreme Pareto” situation is not true)

## Try code generation, giving the compiler more knowledge

The lookup tables are actually known when the program starts, so by using the JIT we could be able to generate optimized code with this information.

# Results - 4 Nodes - VCABM

| | | Time | | | Allocations | | |
|---|---|---|---|---|---|---|---|
| Tot / % measured: | | 1655s / 99.9% | | | 100GiB / 98.2% | | |
| Section | ncalls | time | %tot | avg | alloc | %tot | avg |
| total solver | 1 | 1653s | 100.0% | 1653s | 97.7GiB | 100.0% | 97.7GiB |
| getDeriv! | 278 | 1083s | 65.5% | 3.90s | 77.9GiB | 79.7% | 287MiB |
| getXBubble! | 278 | 1026s | 62.1% | 3.69s | 145MiB | 0.1% | 535KiB |
| partition | 278 | 952s | 57.6% | 3.42s | 139MiB | 0.1% | 512KiB |
| communication | 278 | 67.3s | 4.1% | 242ms | 3.33MiB | 0.0% | 12.2KiB |
| get_ranges | 278 | 11.7ms | 0.0% | 42.2us | 2.64MiB | 0.0% | 9.72KiB |
| rebuildStateStruct! | 278 | 29.8s | 1.8% | 107ms | 77.7GiB | 79.5% | 286MiB |
| repackStateVector! | 278 | 12.7s | 0.8% | 45.6ms | 447KiB | 0.0% | 1.61KiB |
| addToVertexFromBubble! | 278 | 5.96s | 0.4% | 21.5ms | 12.1MiB | 0.0% | 44.6KiB |
| symmetrizeBubble! | 278 | 2.27s | 0.1% | 8.17ms | 24.2MiB | 0.0% | 89.2KiB |
| get_Self_Energy! | 278 | 349ms | 0.0% | 1.25ms | 33.4MiB | 0.0% | 123KiB |
| symmetrizeVertex! | 278 | 159ms | 0.0% | 572us | 15.3MiB | 0.0% | 56.5KiB |
| getDFint! | 278 | 1.55ms | 0.0% | 5.59us | 0.00B | 0.0% | 0.00B |
| workspace | 278 | 203us | 0.0% | 731ns | 0.00B | 0.0% | 0.00B |
| setup | 278 | 30.1us | 0.0% | 108ns | 0.00B | 0.0% | 0.00B |