# ACH Meeting November 2025

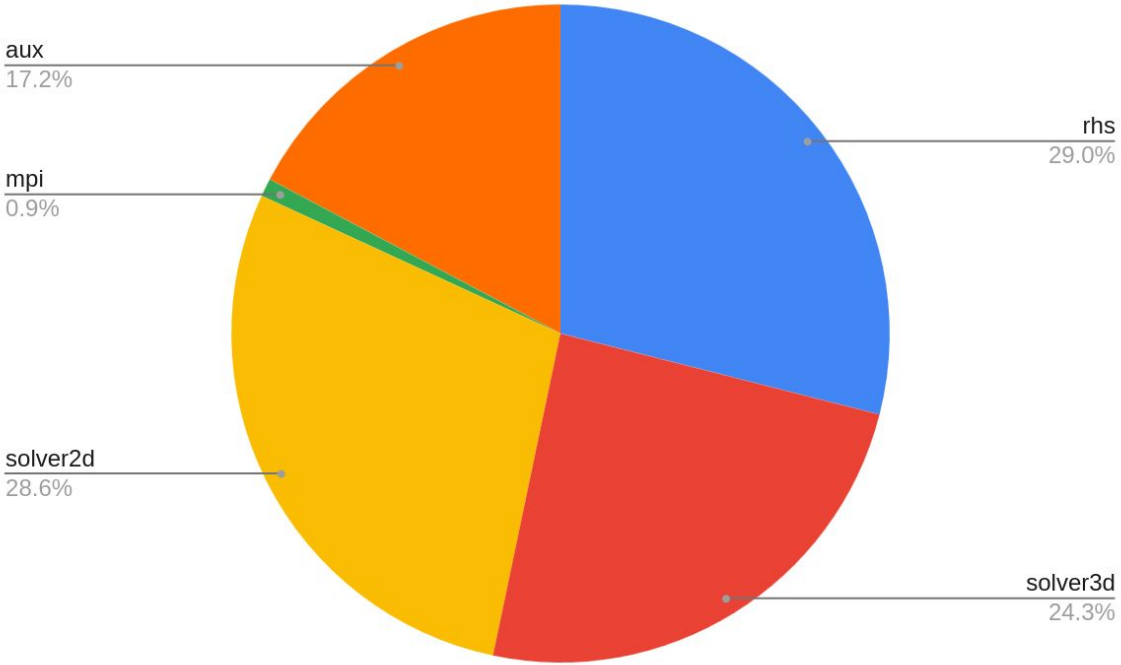
Nicola Varini

# GRILLIX

Nicola Varini

Andreas Stegmeir

# GRILLIX




- The solver2d is performed by parallax:
  - Collaboration for GPU porting
- The **solver3d** will be ported on GPU by ACH
- The RHS will be ported to GPU last.

# The 3D solver applied to heat flux solver

We need to solve

$$\lambda u - \xi \mathbf{P} c \mathbf{Q} u = b$$

In its most basic form, the provided matvec routine does the following:
(see GRILLIX: src/solver_aligned3d/solve_aligned3d_s.f90)

- Send/receive u to/from rank+1/rank-1 **[MPI COMMUNICATION]**
- Multiply Q*u blockwise →heat flux q **[SPMV OPERATION]**
- Send/receive q to/from rank-1/rank+1 **[MPI COMMUNICATION]**
- Multiply P*q blockwise **[SPMV OPERATION]**

# The 3D solver code

```
        ! Parallel gradient
        if ( (tstep_implicit_level == 0) &
            .or.(tstep_implicit_level == 1) ) then

            call perf_start('        pardif_comp')
#ifdef ENABLE_CUDA
            !call debug_device(rank)
            call map%par_grad_cuda(var%vals, var%hfwd, dvar_dpar%vals, 1)
#else
            !$omp parallel default(none) shared(map, var, dvar_dpar)
            call map%par_grad(var%vals, var%hfwd, dvar_dpar%vals)
            !$omp end parallel
#endif
            call perf_stop('        pardif_comp')

            ! Parallel divergence
            call perf_start('        pardif_mpi')
            call dvar_dpar%start_comm_halos(comm_handler)
            call dvar_dpar%finalize_comm_halos(comm_handler)
            call perf_stop('        pardif_mpi')
            call perf_start('        pardif_comp')
#ifdef ENABLE_CUDA
            call map%par_divb_cuda(dvar_dpar%vals, dvar_dpar%hbwd, d2var_dpar2)
#else
            !$omp parallel default(none) shared(map, dvar_dpar, d2var_dpar2)
            call map%par_divb(dvar_dpar%vals, dvar_dpar%hbwd, d2var_dpar2)
            !$omp end parallel
#endif
            call perf_stop('        pardif_comp')
        endif
```

- Parallel gradient  and parallel divergence are ported in CUDA with the cusparse library.
    - Arrays are passed between Fortran and C++.
- The auxiliary arrays are allocated with cudaMallocManaged for easy interoperability with CPU.
- MPI call in between.

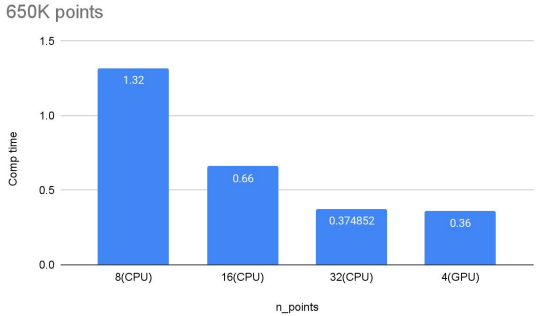
# The 3D solver code

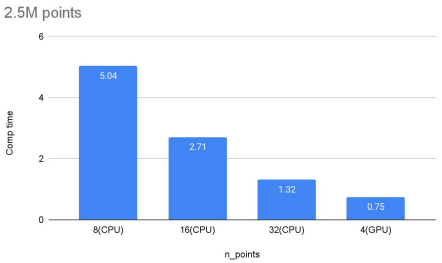
```
    use perf_m, only : perf_start, perf_stop
    class(parallel_map_t), intent(in) :: self
    real(GP), dimension(self%grad_stag_cano_bwd%ncol), intent(in) :: ucano
    real(GP), dimension(self%grad_stag_cano_fwd%ncol), intent(in) :: ucano_fwd
    real(GP), dimension(self%grad_stag_cano_fwd%ndim), intent(out) :: par_grad_u_stag

    call perf_start('pgrad_cuda_copy')
    call copydoubleh2d(self%grad_stag_cano_fwd_cuda%x, ucano_fwd, self%grad_stag_cano_
fwd_cuda%nrows)
    call copydoubleh2d(self%grad_stag_cano_bwd_cuda%x, ucano, self%grad_stag_cano_bwd_
cuda%nrows)
    call setvecvals(self%grad_stag_cano_fwd_cuda%cuda_struct, self%grad_stag_cano_fwd_
cuda%x)
    call setvecvals(self%grad_stag_cano_bwd_cuda%cuda_struct, self%grad_stag_cano_bwd_
cuda%x)
    call perf_stop('pgrad_cuda_copy')
    call perf_start('pgrad_cuda_spmv')
    call spmv_cusparse(self%grad_stag_cano_fwd_cuda%cuda_struct)
    call spmv_cusparse(self%grad_stag_cano_bwd_cuda%cuda_struct)
    call perf_stop('pgrad_cuda_spmv')
    call perf_start('pgrad_cuda_update')
    par_grad_u_stag = self%grad_stag_cano_fwd_cuda%y-self%grad_stag_cano_bwd_cuda%y
    call perf_stop('pgrad_cuda_update')

end subroutine
```

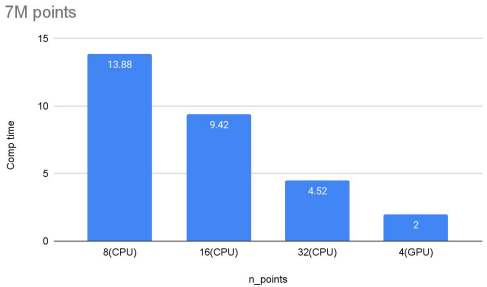
```
    use perf_m, only : perf_start, perf_stop
    class(parallel_map_t), intent(in) :: self
    real(GP), dimension(self%grad_stag_cano_bwd%ncol), intent(in) :: ucano
    real(GP), dimension(self%grad_stag_cano_fwd%ncol), intent(in) :: ucano_fwd
    real(GP), dimension(self%grad_stag_cano_fwd%ndim), intent(out) :: par_grad_u_stag

    call perf_start('pgrad_cuda_copy')
    call copydoubleh2d(self%grad_stag_cano_fwd_cuda%x, ucano_fwd, self%grad_stag_cano_
fwd_cuda%nrows)
    call copydoubleh2d(self%grad_stag_cano_bwd_cuda%x, ucano, self%grad_stag_cano_bwd_
cuda%nrows)
    call setvecvals(self%grad_stag_cano_fwd_cuda%cuda_struct, self%grad_stag_cano_fwd_
cuda%x)
    call setvecvals(self%grad_stag_cano_bwd_cuda%cuda_struct, self%grad_stag_cano_bwd_
cuda%x)
    call perf_stop('pgrad_cuda_copy')
    call perf_start('pgrad_cuda_spmv')
    call spmv_cusparse(self%grad_stag_cano_fwd_cuda%cuda_struct)
    call spmv_cusparse(self%grad_stag_cano_bwd_cuda%cuda_struct)
    call perf_stop('pgrad_cuda_spmv')
    call perf_start('pgrad_cuda_update')
    par_grad_u_stag = self%grad_stag_cano_fwd_cuda%y-self%grad_stag_cano_bwd_cuda%y
    call perf_stop('pgrad_cuda_update')

end subroutine
```

```
extern "C" void spmv_cusparse_(matvec_struct *& mv) {
    cusparseSpMV(
        mv->handle, CUSPARSE_OPERATION_NON_TRANSPOSE,
        &(mv->alpha),mv->matA,mv->vecX,&(mv->beta),mv->vecY, CUDA_R_64F,
        CUSPARSE_SPMV_CSR_ALG1, mv->dBuffer);

    cudaDeviceSynchronize();
```

# Benchmark results




- We measure only the compute time.
- In the current implementation there's a memory transfer operation which should disappear when the remaining part of GRILLIX are ported on GPU.
- At scale the GPU is ~2 times faster compared to CPU(32 threads)

# Feltor

Nicola Varini

Matthias Wiesemberg

# Miniapp

3D Elliptic Operator Implementation in the FELTOR Framework

## Purpose

- Implements a 3D elliptic operator solver (boundary-handling version) using FELTOR's discontinuous Galerkin (dG) framework.

## Key Functionalities

• Use the same multigrid solver as in Feltor.
• Applies this operator to vectors (right-hand side).
• Interfaces with FELTOR's grid, weights, and solver modules.

## Inputs & Outputs

Inputs: Grid geometry, σ field, boundary condition types.
Outputs: Operator object, solution vector (φ).

# Miniapp - benchmark

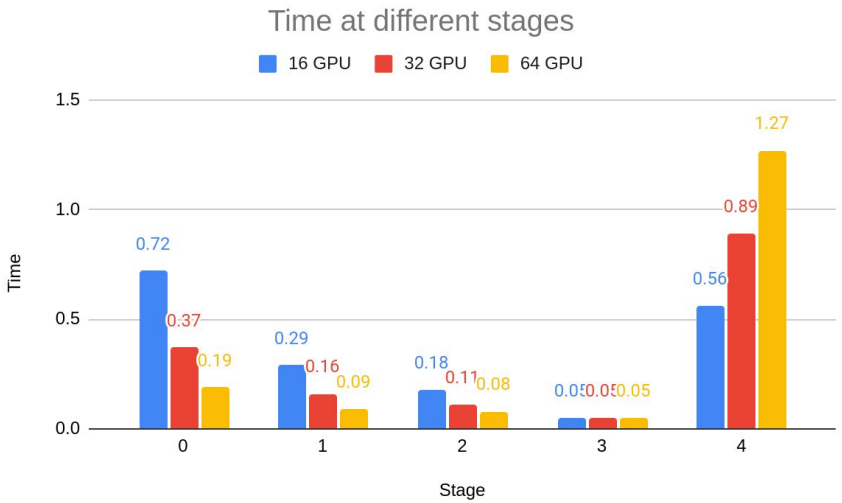
- Goal: to study the performance of the elliptic solver
  - It is interesting to test the scaling in the poloidal plane

- Benchmark size:
  - Nx=2048, Ny=4096, Nz=32
  - 5 stage multigrid

- Strong scaling and profiling on Pitagora@GPU

```
cmd = [
        "nsys", "stats", "-r", val, "--filter-nvtx", "stage"+str(stage), "--format",
        "json","--output",".","--force-overwrite","true",filename
]
val = {
 "mpi": "mpi_event_sum",
 "cuda": "cuda_gpu_sum"
 }
```

# Solver profiling - Native implementation

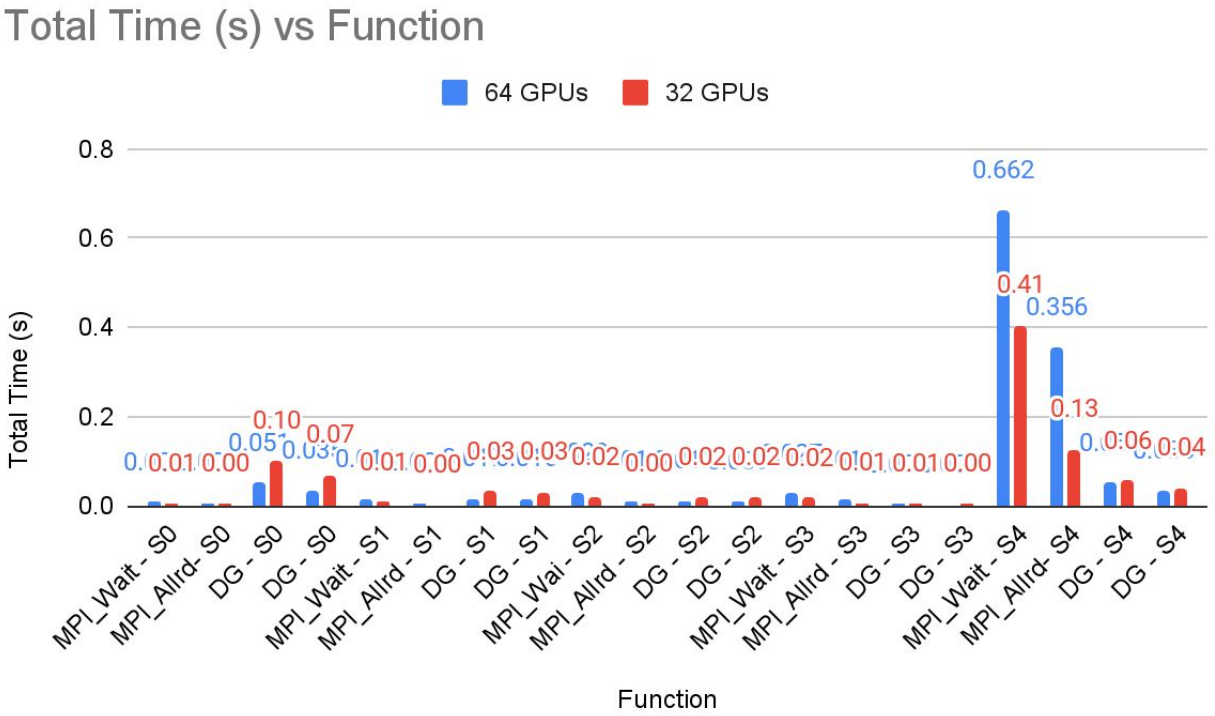



- Good scaling for Stage 0-2(finest)
- Stage 3 and 4 are coarser and the scaling is not good
- Stage 4 takes many iterations
  - Artifact of the simulation

| nodes | TTS |
|---|---|
| 16 | 1.85 |
| 32 | 1.74 |
| 64 | 1.99 |

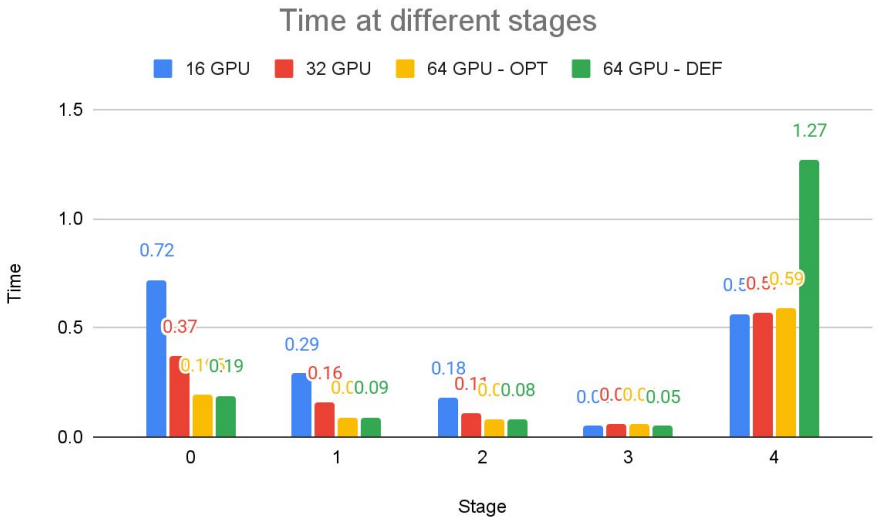
| #TASK | Stage | Time | Iteration |
|---|---|---|---|
| 0 | 0 | 0.72 | 23 |
| 1 | 1 | 0.29 | 40 |
| 2 | 2 | 0.18 | 90 |
| 3 | 3 | 0.05 | 90 |
| 4 | 4 | 0.56 | 2200 |
| 0 | 0 | 0.37 | 23 |
| 1 | 1 | 0.16 | 40 |
| 2 | 2 | 0.11 | 90 |
| 3 | 3 | 0.05 | 90 |
| 4 | 4 | 0.89 | 2200 |
| 0 | 0 | 0.19 | 23 |
| 1 | 1 | 0.09 | 40 |
| 2 | 2 | 0.08 | 90 |
| 3 | 3 | 0.05 | 90 |
| 4 | 4 | 1.27 | 2200 |

# Profiling different stages




- By using nsys it is possible to extract how much time is spent in MPI and CUDA
- On Stage 4 the communication time explodes
  - Lots of iterations
- Still, it is instructive to explore mitigation strategies

# Solver optimization




| nodes | TTS - OPT MPI | TTS - DEF MPI |
|---|---|---|
| 16 | 1.85 | 1.85 |
| 32 | 1.32 | 1.74 |
| 64 | 1.15 | 1.99 |

| #TASK | Stage | Time - Opt | Time - Def |
|---|---|---|---|
| 16 | 0 | 0.72 | 0.72 |
| 16 | 1 | 0.29 | 0.29 |
| 16 | 2 | 0.18 | 0.18 |
| 16 | 3 | 0.05 | 0.05 |
| 16 | 4 | 0.56 | 0.56 |
| 32 | 0 | 0.37 | 0.37 |
| 32 | 1 | 0.16 | 0.16 |
| 32 | 2 | 0.11 | 0.11 |
| 32 | 3 | 0.06 | 0.05 |
| 32 | 4 | 0.57 | 0.89 |
| 64 | 0 | 0.195 | 0.19 |
| 64 | 1 | 0.09 | 0.09 |
| 64 | 2 | 0.08 | 0.08 |
| 64 | 3 | 0.06 | 0.05 |
| 64 | 4 | 0.59 | 1.27 |

- For level 3 and 4 MPI_Gather on Rank 0
- Next:
  - Experiment with NCCL/NVSHMEM
  - Benchmark a realistic feltor simulation

# GBS

Nicola Varini
Micol Bassanini

# Two-fluid turbulent plasma model

**No electromagnetic effects**

**Density**

$$\frac{\partial n}{\partial t} = -\frac{\rho_*^{-1}}{B}[\phi, n] + C(p_e, \phi, n) - \nabla_\parallel(nV_{\parallel e}) + D_n\nabla_\perp^2 n + s_n$$

**Vorticity**

$$\frac{\partial \Omega}{\partial t} = -\frac{\rho_*^{-1}}{B}\nabla\cdot[\phi, n\nabla_\perp\phi] - \nabla\cdot(V_{\parallel i}\nabla_\parallel(n\nabla_\perp\phi)) + B^2\nabla_\parallel(j_\parallel) + D_\Omega\nabla_\perp^2\Omega + C(p_e, p_i) + s_\Omega$$

**Electron Parallel Velocity**

$$\frac{\partial V_{\parallel e}}{\partial t} = -\frac{\rho_*^{-1}}{B}[\phi, V_{\parallel e}] - V_{\parallel e}\nabla_\parallel V_{\parallel e} + \frac{m_i}{m_e}\left(\nu j_\parallel + \nabla_\parallel\phi - \frac{1}{n}\nabla_\parallel p_e - 0.71\nabla_\parallel T_e\right) + \frac{4}{3n}\eta_{0e}\nabla_\parallel^2 V_{\parallel e} + s_{V_\parallel}$$

**How can we efficiently integrate this system in time?**

**Ion Parallel Velocity**

$$\frac{\partial V_{\parallel i}}{\partial t} = -\frac{\rho_*^{-1}}{B}[\phi, V_{\parallel i}] - V_{\parallel i}\nabla_\parallel V_{\parallel i} - \frac{1}{n}\nabla_\parallel(p_e + \tau p_i) + \frac{4}{3n}\eta_{0i}\nabla_\parallel^2 V_{\parallel i} + D_{V_{\parallel i}}\nabla_\perp^2 V_{\parallel i} + s_{V_{\parallel i}}$$

**Electron Temperature**

$$\frac{\partial T_e}{\partial t} = -\frac{\rho_*^{-1}}{B}[\phi, T_e] - V_{\parallel e}\nabla_\parallel T_e + \frac{2}{3}T_e\left[0.71\frac{\nabla_\parallel j_\parallel}{n} - \nabla_\parallel V_{\parallel e}\right] + C(T_e, n, \phi) + \nabla_\parallel(\chi_{\parallel e}\nabla_\parallel T_e) + s_{T_e}$$

**Ions Temperature**

$$\frac{\partial T_i}{\partial t} = -\frac{\rho_*^{-1}}{B}[\phi, T_i] - V_{\parallel i}\nabla_\parallel T_i + \frac{2}{3}T_i\left[j_\parallel\frac{\nabla_\parallel n}{n^2} - \nabla_\parallel V_{\parallel e}\right] + C(T_i, T_e, n, \phi) + \nabla_\parallel(\chi_{\parallel i}\nabla_\parallel T_i) + s_{T_i}$$

**Electrostatic Potential**

$$\nabla\cdot(n\nabla_\perp\phi) = \Omega - \tau\nabla_\perp p_i \quad \text{Algebraic constraint}$$

EPFL

# Different Time integration schemes

## Explicit Time integration schemes:

- **Pros:**
  - Easy to implement
  - Each time step is computationally fast
  - Easy parallelizable
- **Cons:**
  - CFL condition on the $\Delta t$

## Implicit Time integration schemes:

- **Pros:**
  - No CFL condition (most of them unconditionally stable)
- **Cons:**
  - Expensive per step
  - Complex implementation
  - Harder to parallelize

## Implicit-Explicit (IMEX) time integration scheme:

- **Pros:**
  - Balances stability & efficiency: non-stiff terms are treated explicitly and stiff terms treated implicitly
- **Cons:**
  - Prior knowledge of the physics
  - Still needs implicit solver (implementation complexity)

# Treatment of the Time-Scale separation

Considering $\mathbf{q} = [n, \Omega, V_{\|e}]$ and $z = \phi$, the system is of the form:

$$\begin{aligned} \frac{\partial \mathbf{q}}{\partial t} &= \mathbf{F}(\mathbf{q}, z) \\ 0 &= g(\mathbf{q}, z) \end{aligned}$$

Differential Algebraic equation of 1 index DAE-1

where $\frac{\partial g}{\partial z}$ has a bounded inverse.

$$\mathbf{F}(\mathbf{q}, z) = \mathbf{F}_S(\mathbf{q}, z) + \mathbf{F}_F(\mathbf{q}, z)$$

$\mathbf{F}_S$ are the slow components and are **integrated explicitly**.

$\mathbf{F}_F$ are the fast components and are **integrated implicitly**.

## Implicit-Explicit (IMEX) time integration scheme:

- **Pros:**
  - Balances stability & efficiency: non-stiff terms are treated explicitly and stiff terms treated implicitly
- **Cons:**
  - Prior knowledge of the physics
  - Still needs implicit solver (implementation complexity)

# IMEX-RK for DAE-1: complete implementation



**At every internal stage:**

RHS

Computation of the rhs: $\mathrm{rhs}_{T_e}$, $\mathrm{rhs}_{T_i}$, $\mathrm{rhs}_{V_{\parallel i}}$, $\mathrm{rhs}_{\Omega}$, $\mathrm{rhs}_{V_{\parallel e}}$, $\mathrm{rhs}_{\phi}$

$$\mathrm{rhs}_g = g^n + \Delta t \sum_{j=1}^{i-1} a_{ij} f_{g,S} + \Delta t \sum_{j=1}^{i-1} \bar{a}_{ij} f_{g,F}$$

$T_{e,i}$

$$\left[\mathbf{I} - \bar{a}_{ii}\Delta t \chi_e \frac{1}{T_e^{(i-1)}} \nabla_\parallel \left(T_e^{(i-1)} \nabla_\parallel \bullet\right)\right] \left[t_e^{(i)}\right] = \left[\mathrm{rhs}_{T_e}\right] \qquad T_e^{(i)} = \exp t_e^{(i)}$$

$$\left[\mathbf{I} - \bar{a}_{ii}\Delta t \chi_i \frac{1}{T_i^{(i-1)}} \nabla_\parallel \left(T_i^{(i-1)} \nabla_\parallel \bullet\right)\right] \left[t_i^{(i)}\right] = \left[\mathrm{rhs}_{T_i}\right] \qquad T_i^{(i)} = \exp t_i^{(i)}$$

$V_{\parallel i}$

$$\left[\mathbf{I} - \bar{a}_{ii}\Delta t \frac{4}{3n^{(i)}} \nabla_\parallel^2\right] \left[V_{\parallel i}^{(i)}\right] = \left[\mathrm{rhs}_{V_{\parallel i}}\right]$$

$n$

$$\theta^{(i)} = \theta^n + \Delta t \sum_{j=1}^{i-1} a_{ij} f_{n,S} \qquad n^{(i)} = \exp \theta^{(i)}$$

$\Omega, V_{\parallel e}, \Phi$

$$\begin{bmatrix} \mathbf{I} & \bar{a}_{ii}\Delta t n^{(i)} \nabla_\parallel & \mathbf{0} \\ \mathbf{0} & \mathbf{I} - \bar{a}_{ii}\Delta t \frac{4}{3n^{(i)}} \nabla_\parallel^2 & -\bar{a}_{ii}\Delta t \frac{m_i}{m_e} \nabla_\parallel \\ \mathbf{I} & \mathbf{0} & -\nabla \cdot \left(n^{(i)} \nabla_\perp\right) \end{bmatrix} \begin{bmatrix} \Omega^{(i)} \\ V_{\parallel e}^{(i)} \\ \phi^{(i)} \end{bmatrix} = \begin{bmatrix} \mathrm{rhs}_{\Omega} \\ \mathrm{rhs}_{V_{\parallel e}} \\ \mathrm{rhs}_{\phi} \end{bmatrix}$$

# IMEX Implicit Systems Summary

**Linear System that couples $V_{\parallel e}, \Omega, \Phi$:**

Solved using FGMRES (rel. tol $10^{-8}$) with

**Physics-based preconditioner P**
To compute $P^{-1}$ we need to invert

- **Wave matrix** — Solved with PGMRES (rel. tol $10^{-3}$) using AMG
- **Diffusion matrix** — Solved with PGMRES (rel. tol $10^{-3}$) using AMG

**Linear System for $V_{\parallel i}$:**

Solved using FGMRES (rel. tol $10^{-10}$) with **Hypre BoomerAMG** preconditioner

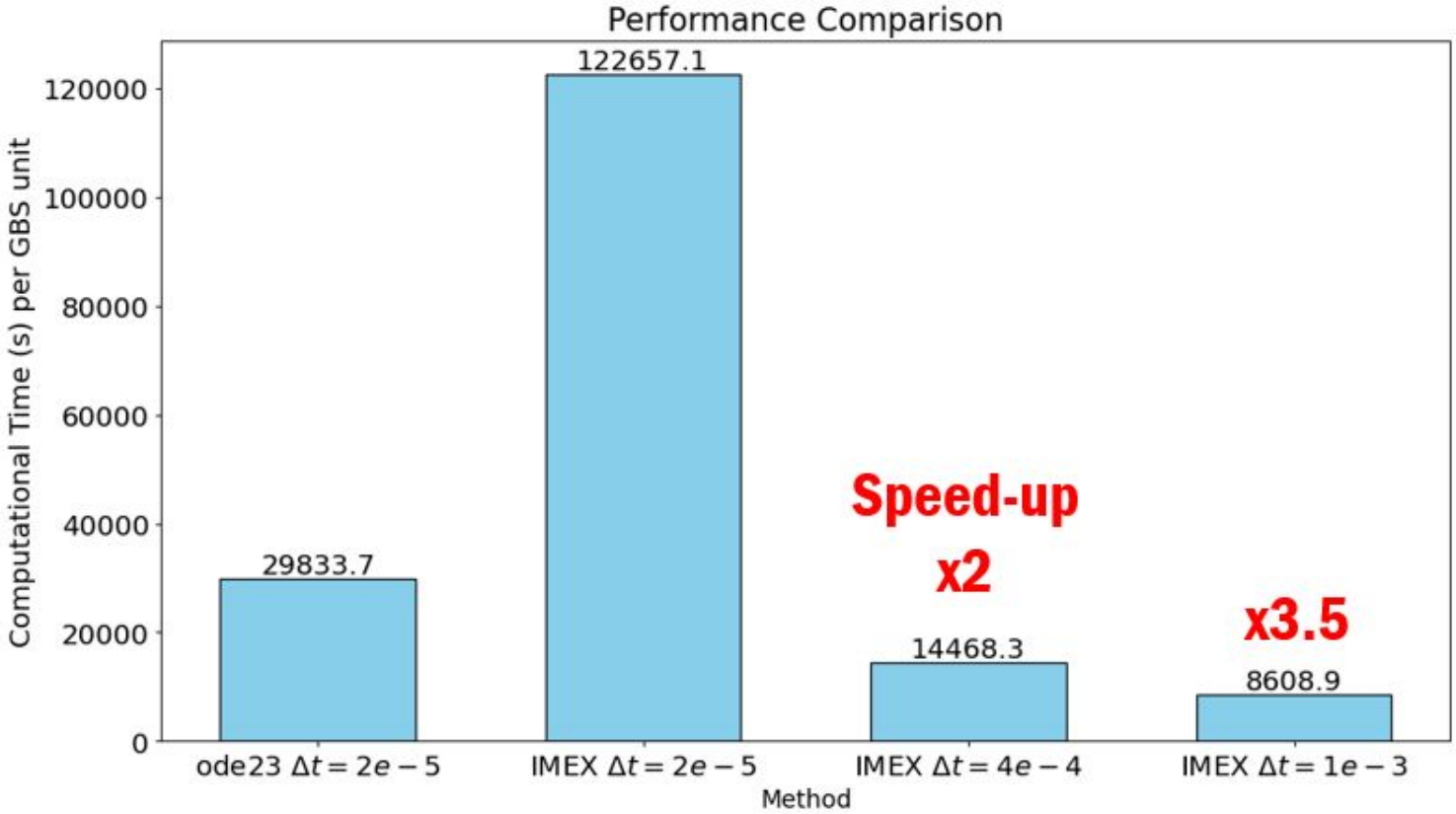
**Linear System for $T_e$:**

Solved using FGMRES (rel. tol $10^{-10}$) with **Hypre BoomerAMG** preconditioner

**Linear System for $T_i$:**

Solved using FGMRES (rel. tol $10^{-8}$) with **Hypre BoomerAMG** preconditioner

# Improved computational efficiency




Test conducted with 4 nodes, 128 tasks per node. LUMI architecture