Performance analysis & optimization of DLR High-Performance Computing codes

Immo Huismann, Ronny Tschüter, Jana Gericke, and Michael Wagner

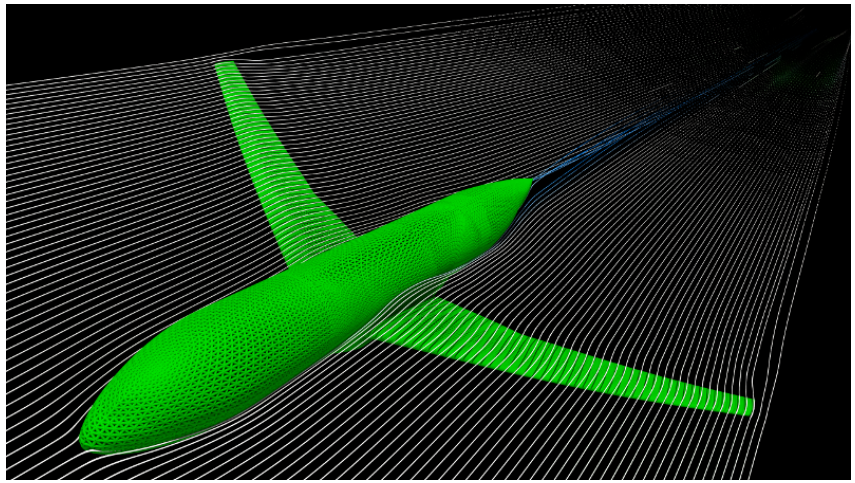
DLR German Aerospace Center
Software Methods for Product Virtualization
High-Performance Computing
Dresden

9th of June, 2022

Simulations become larger

- 2006 A380[1]: $\approx 48 \cdot 10^6$ grid points
  - landing at low speeds, $k - \omega$ model
- 2020 XRF1: $\approx 105 \cdot 10^6$ elements
- 2020 industry: $\geq 300 \cdot 10^6$ elements
- goal: same turnaround time

$\Rightarrow$ faster hardware and software needed

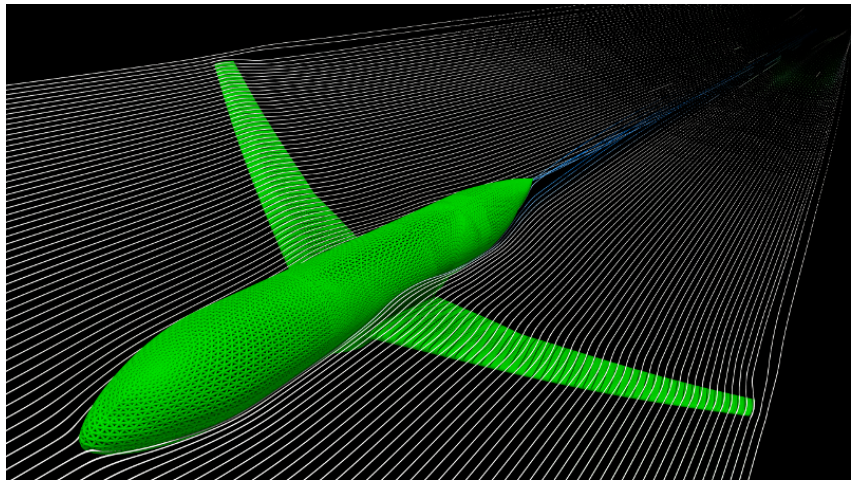


[1] The DLR TAU-code: recent applications in research and industry. D. Schwammborn et. al., ECCOMAS 2006

Simulations become larger

- 2006 A380[1]: $\approx 48 \cdot 10^6$ grid points
  - landing at low speeds, $k - \omega$ model
- 2020 XRF1: $\approx 105 \cdot 10^6$ elements
- 2020 industry: $\geq 300 \cdot 10^6$ elements
- goal: same turnaround time

$\Rightarrow$ faster hardware and software needed



HPC systems become larger

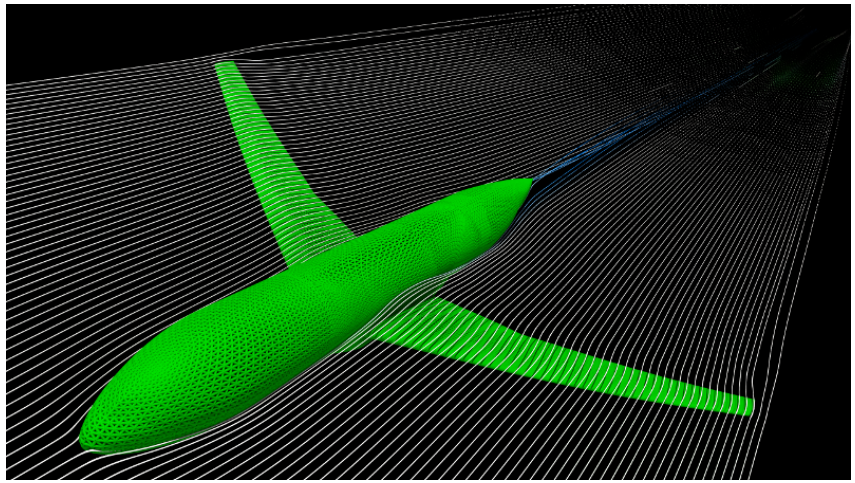
- 1999 NEC SX5: 12 cores
- 2007 Opteron cluster: 128 cores
  - More cores & higher frequency
- 2020 CARA: 145.920 cores
  - Only more cores

[1] The DLR TAU-code: recent applications in research and industry. D. Schwammborn et. al., ECCOMAS 2006

Simulations become larger

- 2006 A380[1]: $\approx 48 \cdot 10^6$ grid points
  - landing at low speeds, $k - \omega$ model
- 2020 XRF1: $\approx 105 \cdot 10^6$ elements
- 2020 industry: $\geq 300 \cdot 10^6$ elements
- goal: same turnaround time

$\Rightarrow$ faster hardware and software needed



HPC systems become larger

- 1999 NEC SX5: 12 cores
- 2007 Opteron cluster: 128 cores
  - More cores & higher frequency
- 2020 CARA: 145.920 cores
  - Only more cores

High performance via parallelism

- AMDAHL'S law for speedup S
  - 99.0 % of runtime parallel $\Rightarrow S \leq 100$
  - 99.9 % of runtime parallel $\Rightarrow S \leq 1000$

$\Rightarrow$ Pinpointing wasted cycles needed

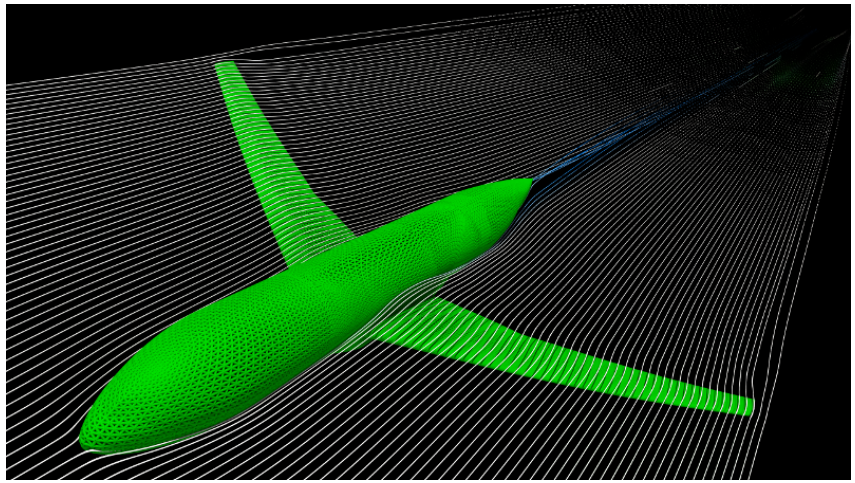
$\Rightarrow$ **In-depth look at computation & communication required**

[1] The DLR TAU-code: recent applications in research and industry. D. Schwammborn et. al., ECCOMAS 2006

Simulations become larger

- 2006 A380[1]: $\approx 48 \cdot 10^6$ grid points
  - landing at low speeds, $k - \omega$ model
- 2020 XRF1: $\approx 105 \cdot 10^6$ elements
- 2020 industry: $\geq 300 \cdot 10^6$ elements
- goal: same turnaround time

$\Rightarrow$ faster hardware and software needed



HPC systems become larger

- 1999 NEC SX5: 12 cores
- 2007 Opteron cluster: 128 cores
  - More cores & higher frequency
- 2020 CARA: 145.920 cores
  - Only more cores

Considered cases

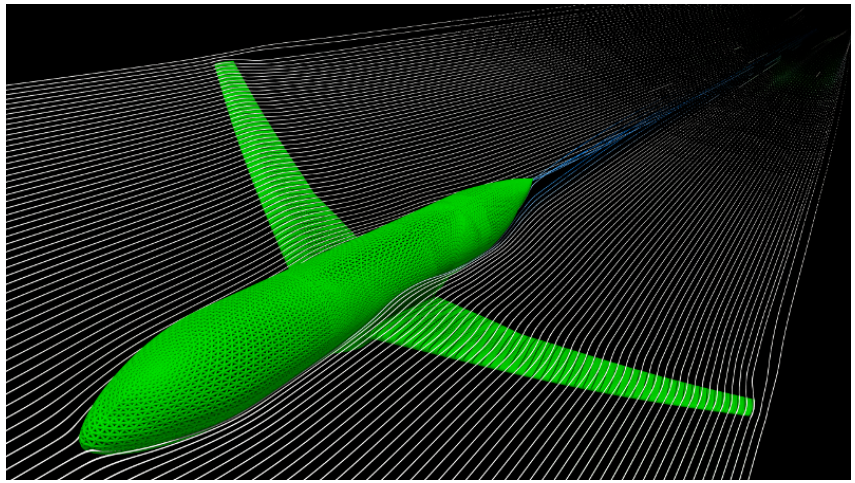
- Musubi
  - Lattice Boltzmann CFD run
- CODA
  - Finite Volume CFD run
- FlowSimulator CFD-CSM pipeline
  - Complex Python-based pipeline

[1] The DLR TAU-code: recent applications in research and industry. D. Schwammborn et. al., ECCOMAS 2006

Simulations become larger

- 2006 A380[1]: $\approx 48 \cdot 10^6$ grid points
  - landing at low speeds, $k - \omega$ model
- 2020 XRF1: $\approx 105 \cdot 10^6$ elements
- 2020 industry: $\geq 300 \cdot 10^6$ elements
- goal: same turnaround time

$\Rightarrow$ faster hardware and software needed



HPC systems become larger

- 1999 NEC SX5: 12 cores
- 2007 Opteron cluster: 128 cores
  - More cores & higher frequency
- 2020 CARA: 145.920 cores
  - Only more cores

Considered cases

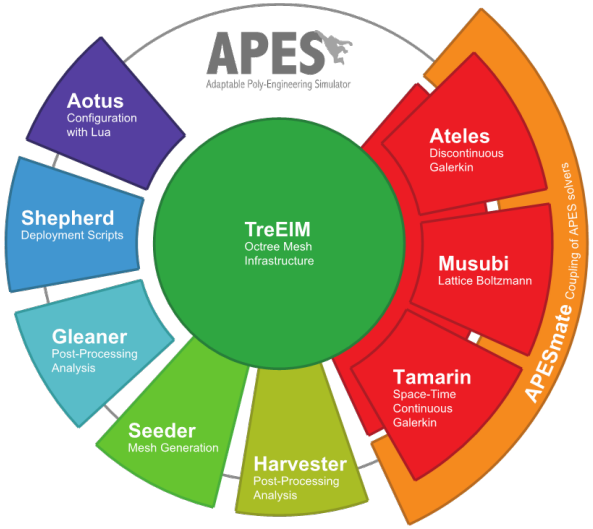
- **Musubi**
  - Lattice Boltzmann CFD run
- **CODA**
  - Finite Volume CFD run
- FlowSimulator CFD-CSM pipeline
  - Complex Python-based pipeline

[1] The DLR TAU-code: recent applications in research and industry. D. Schwammborn et. al., ECCOMAS 2006

Musubi – Open source Lattice Boltzmann solver

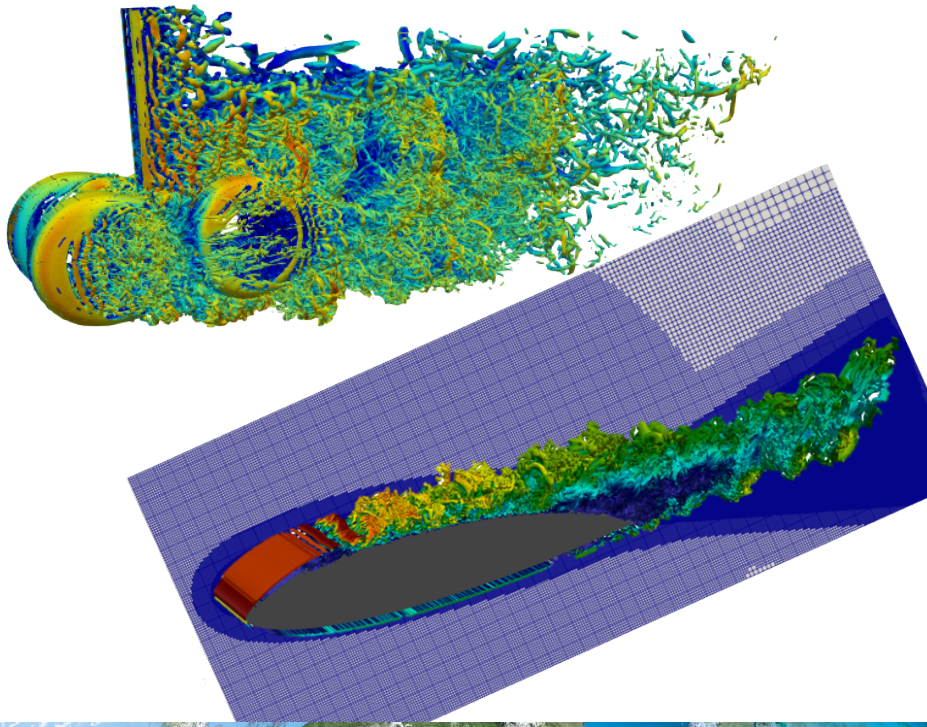
Summary

- Based on the Lattice Boltzmann Method
- Open source solver in the Apes Suite
- Maintained and extended by DLR-SP



Coding

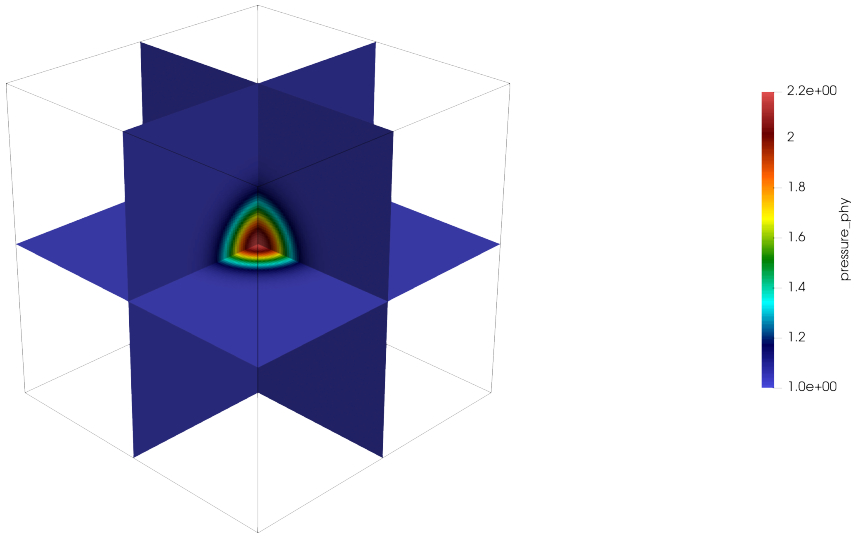
- User scenarios scriptable in Lua
- Solver written in Fortran 2003
- Initially: structured, cartesian grid
  - Adapted in preprocessing step
  - Treated in unstructured fashion
  - Structures can be embedded
- Domain decomposition via MPI



Musubi – Testcase

Gaussian pulse

- Domain $\Omega = [-5, 5]^3$
- Structured mesh using N^3 cells
- BGK collision scheme
- D3Q19 stencil



Explicit time stepping

1. Compute auxilliary variables
2. Compute right-hand side
3. Apply RHS to update solution

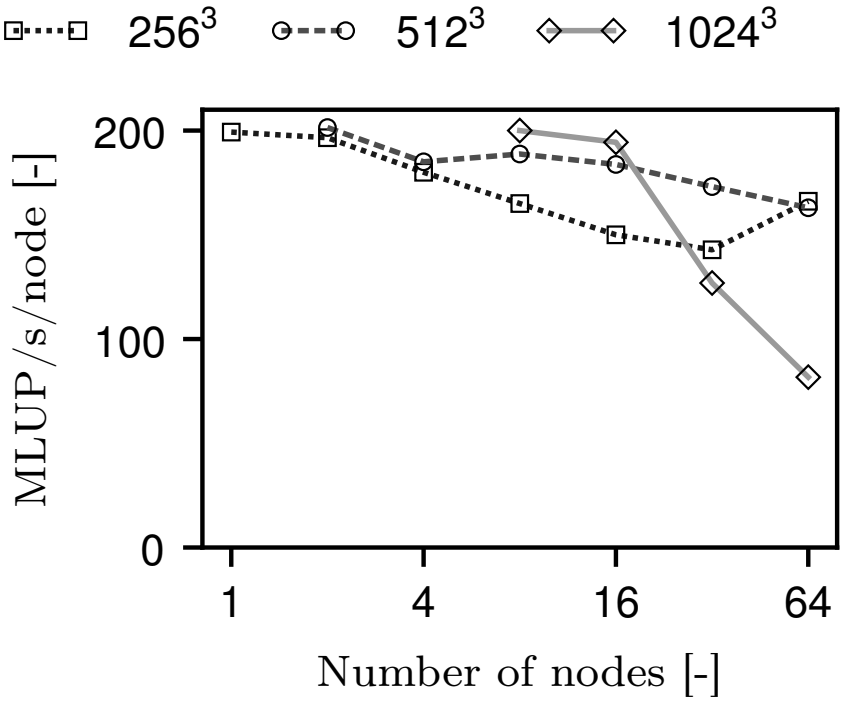
Scalability study

- Cells per direction
  - $N \in \{256^3 \ldots 1024^3\}$
- Number of nodes
  - $1 \ldots 64$
- 64 processes per node

Musubi – Scalability study

Performance measurements

- Testcase ran on CARA
  - 64 processes per node
  - Increasing number of nodes
  - Increasing cells per node
- Runtime of time steps measured
  - Mega Lattice Updates / second shown



Results

- Good scalability even for low # of cells
- Initially increases more cells per node
- Sharp decline for 1024^3 cells

Musubi – Scalability study

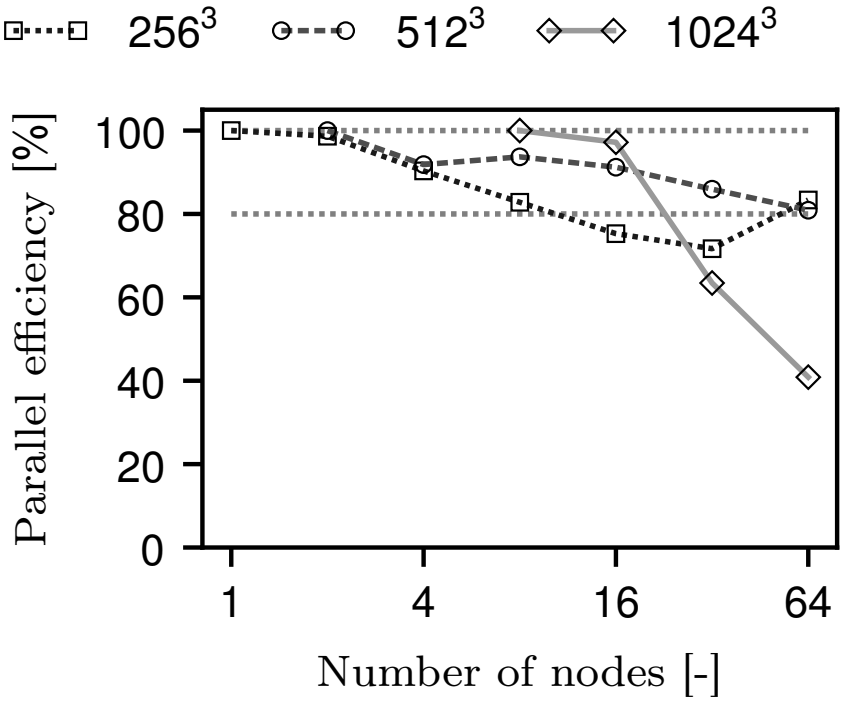
Performance measurements

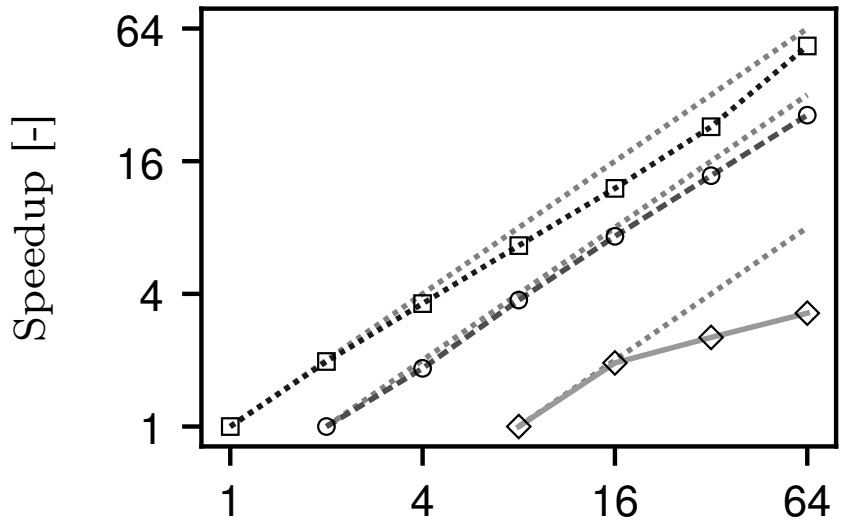
- Testcase ran on CARA
  - 64 processes per node
  - Increasing number of nodes
  - Increasing cells per node
- Runtime of time steps measured
  - Mega Lattice Updates / second shown

Results

- Good scalability even for low # of cells
- Initially increases more cells per node
- Sharp decline for 1024^3 cells

⇒ **Look into communication patterns**





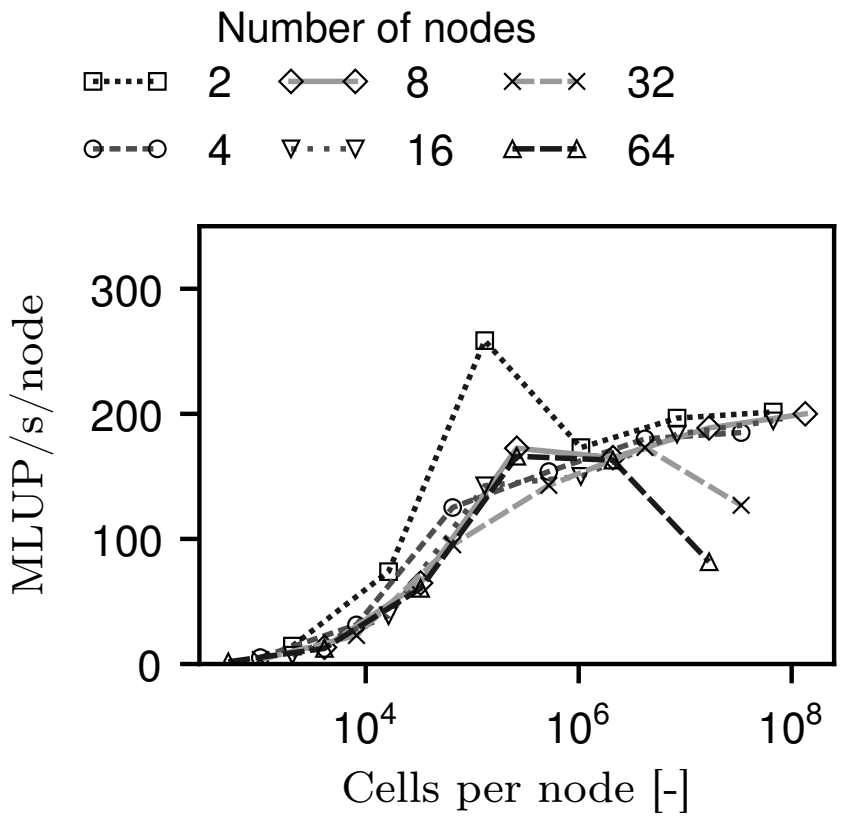
Musubi – Scalability study

Performance measurements

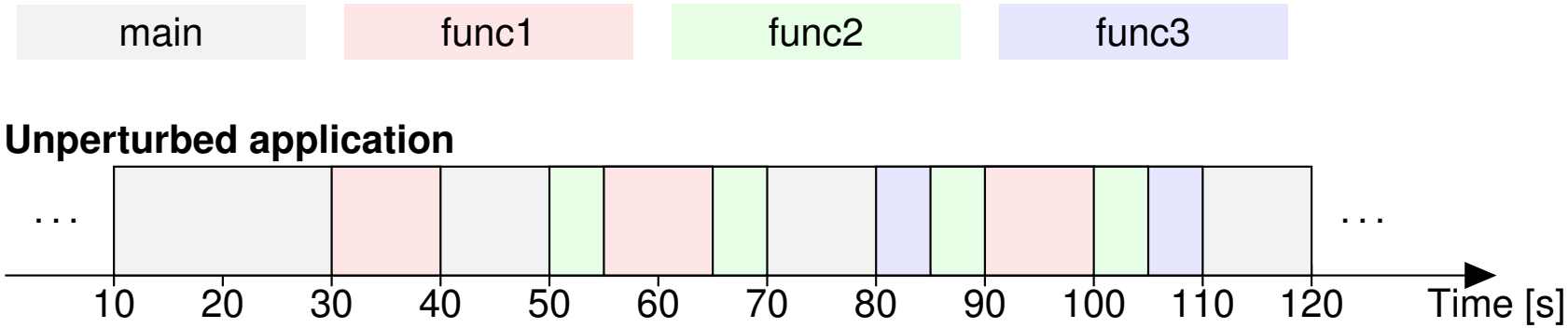
- Testcase ran on CARA
  - 64 processes per node
  - Increasing number of nodes
  - Increasing cells per node
- Runtime of time steps measured
  - Mega Lattice Updates / second shown

Results

- Good scalability even for low # of cells
- Initially increases more cells per node
- Sharp decline for 1024^3 cells
- ⇒ **Look into communication patterns**



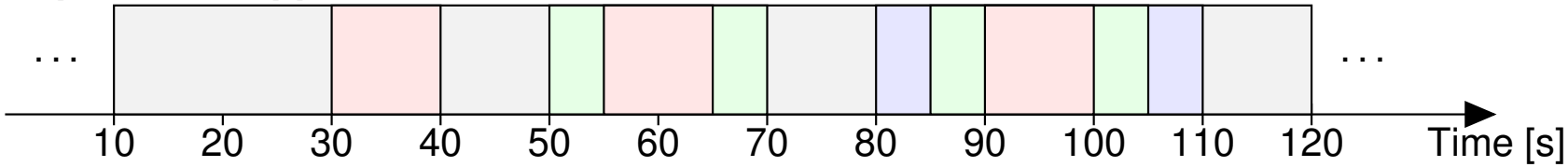
Interludium: Code execution & measurement



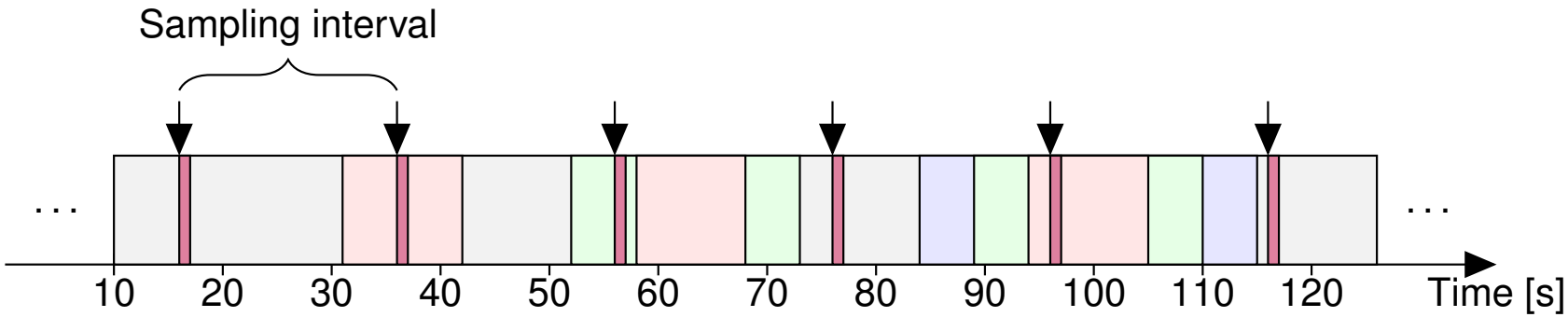
Interludium: Code execution & measurement

main | func1 | func2 | func3 | monitor

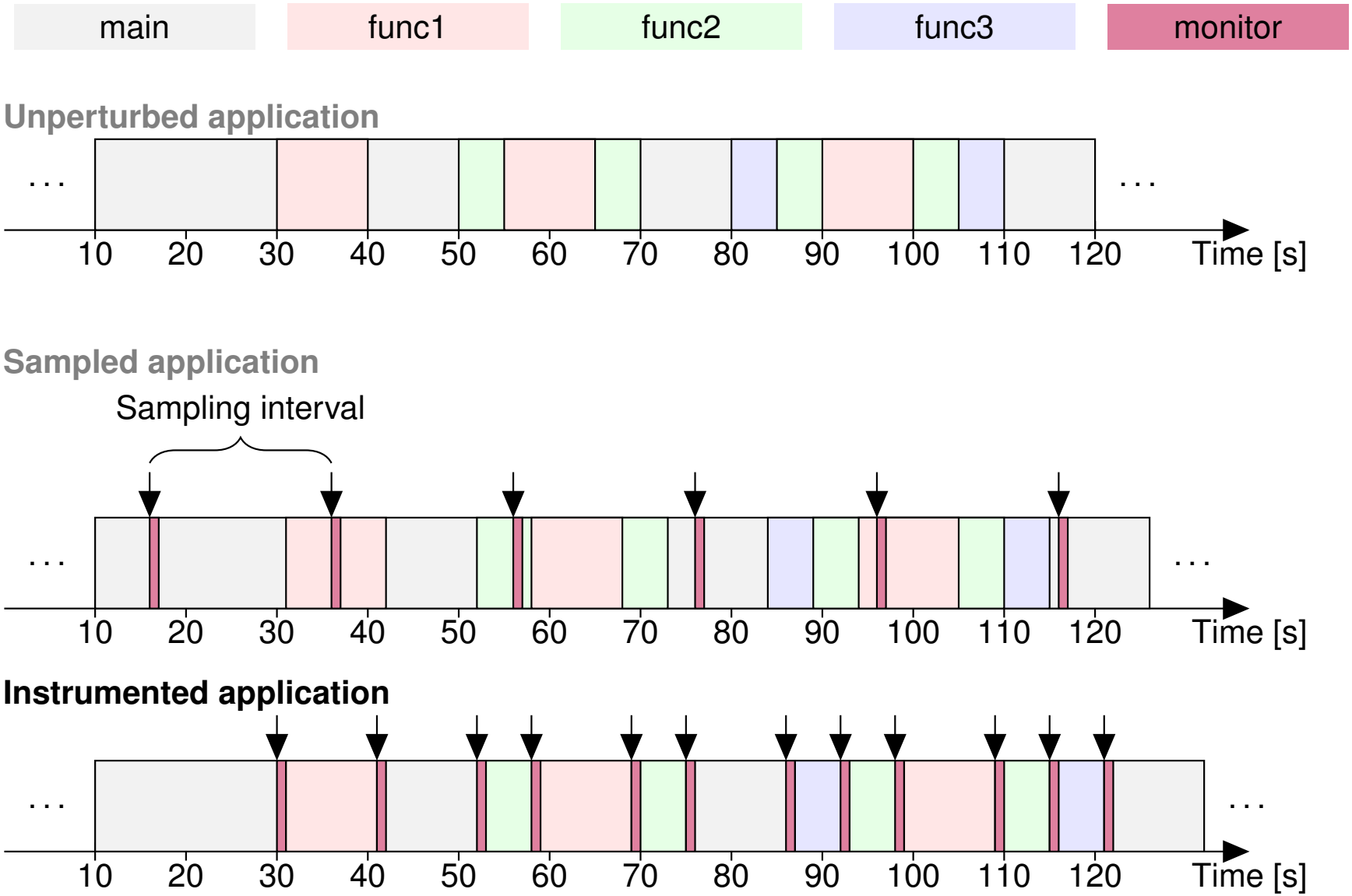
Unperturbed application



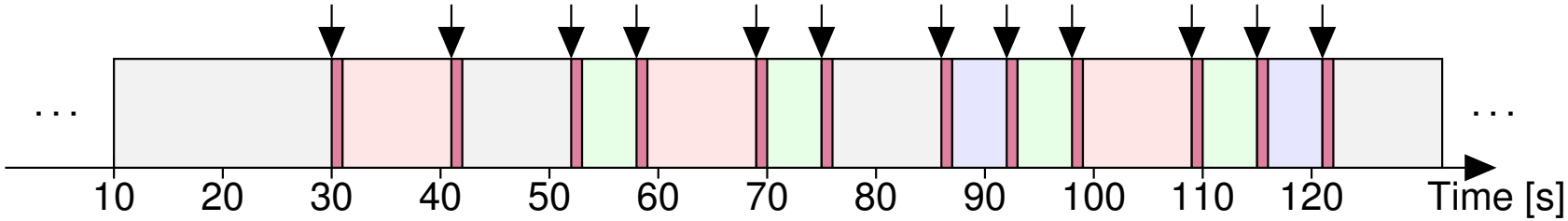
Sampled application



Interludium: Code execution & measurement



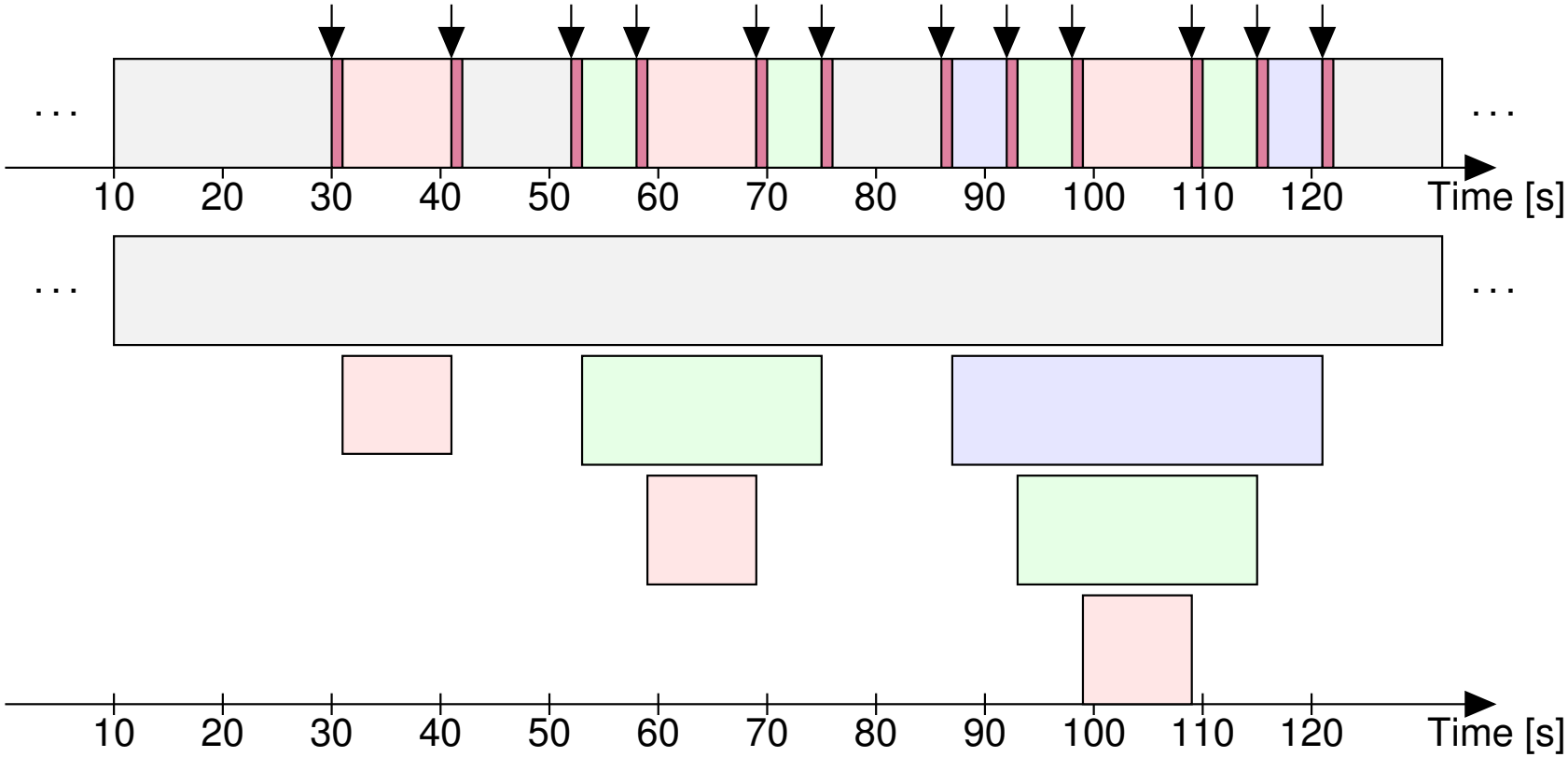
Interludium: Code instrumentation – steps to produce



Steps for instrumentation

1. Identify regions of interest
2. Filter relevant functions
3. Automated instrumentation
4. Run application-relevant testcase

Interludium: Code instrumentation – steps to produce



Steps for instrumentation

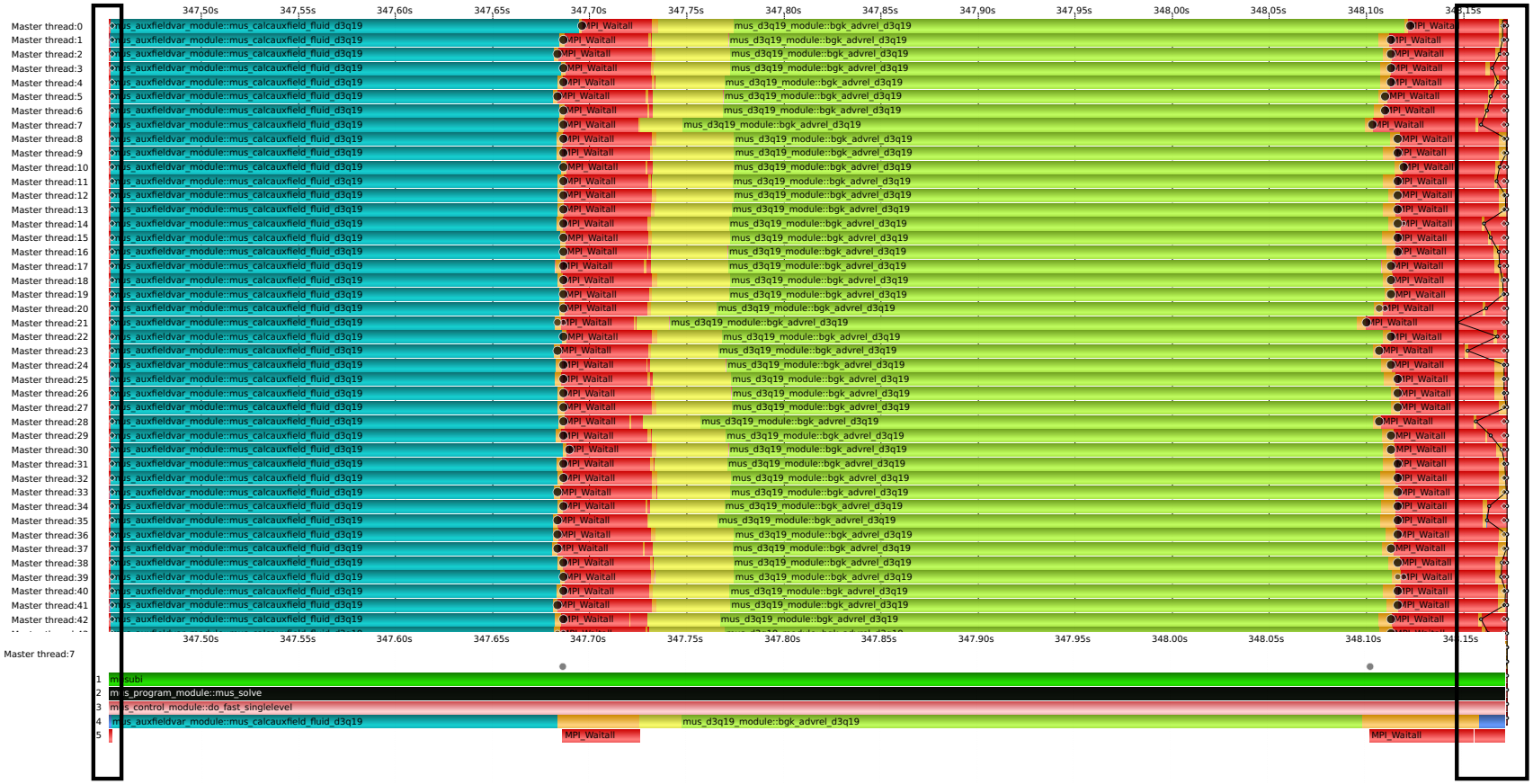
1. Identify regions of interest
2. Filter relevant functions
3. Automated instrumentation
4. Run application-relevant testcase

Result: Timeline

- What was done? when? by whom?
- MPI, OpenMP, pthread, CUDA
- Performance counters (L3 misses. . .)

Musubi – Trace & insights

Musubi – Trace & insights

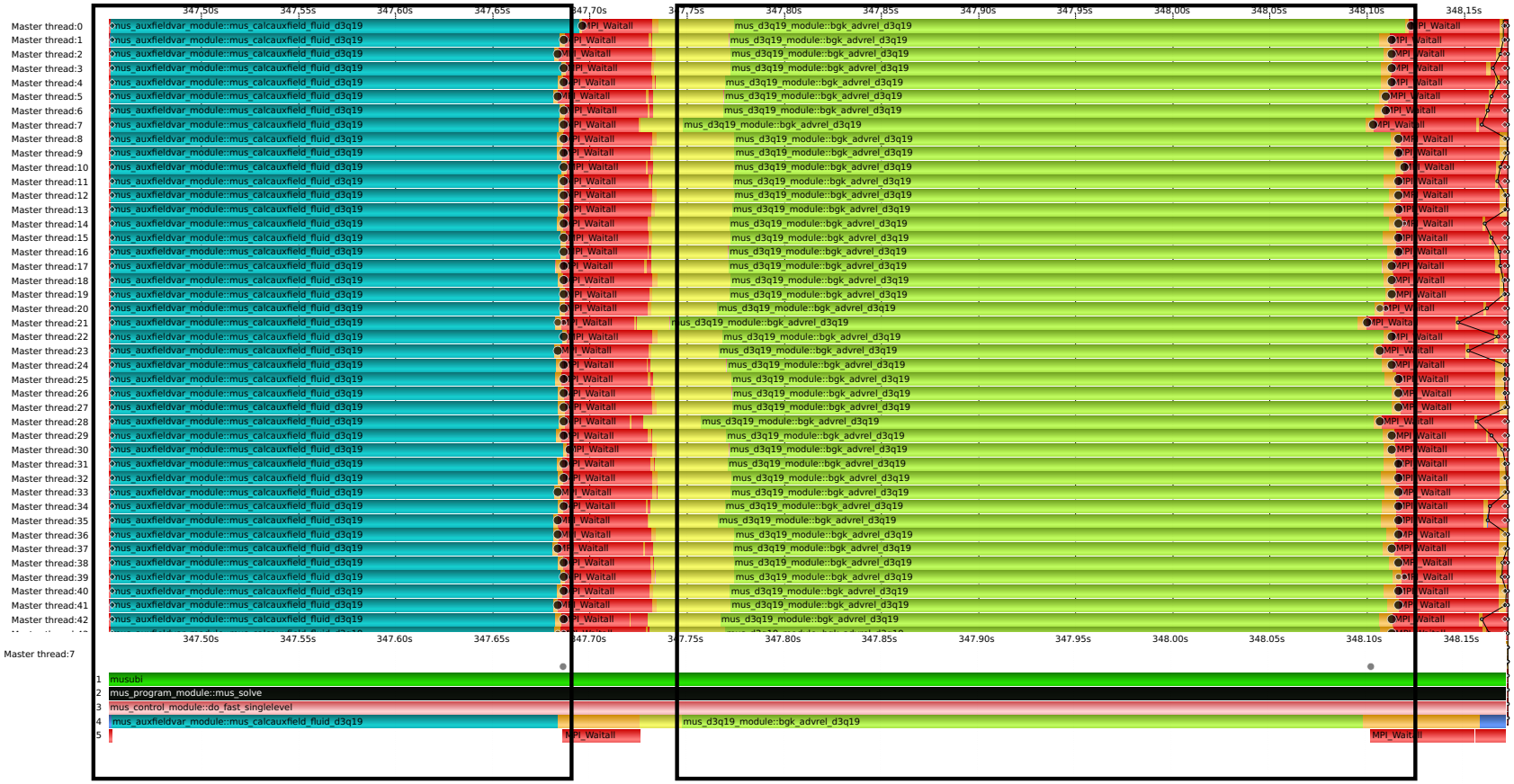


Allreduce

Allreduce

- Processes run in lockstep
  - Synchronization from MPI_Allreduce
  - ⇒ Runtime differences accumulate

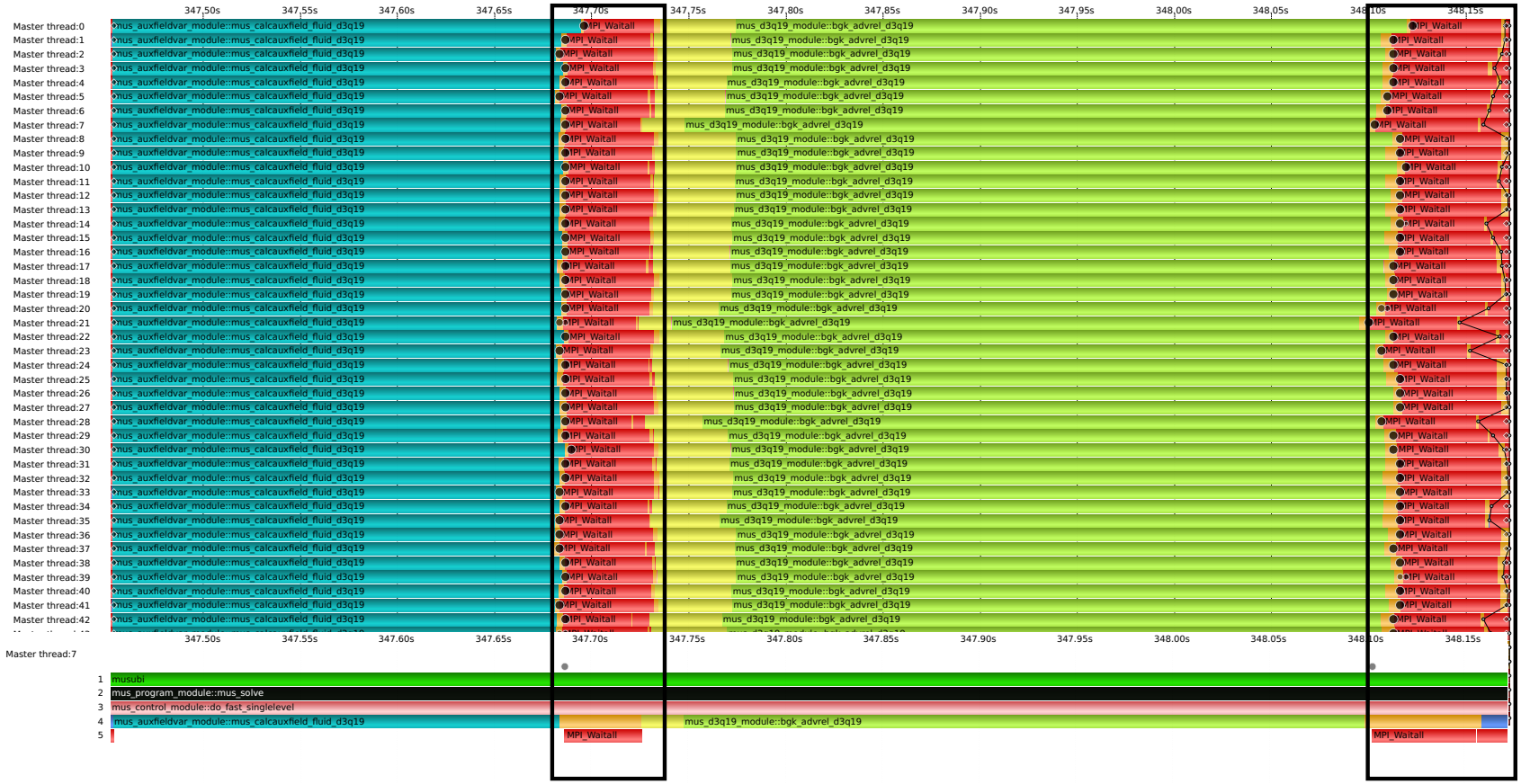
Musubi – Trace & insights



Stencil ρ and $\vec{v}$ | Compute RHS

- Processes run in lockstep
  - Synchronization from MPI_Allreduce
  - $\Rightarrow$ Runtime differences accumulate

Musubi – Trace & insights



- Processes run in lockstep
  - Synchronization from MPI_Allreduce
  - ⇒ Runtime differences accumulate
- No overlap of comm. and comp.
  - MPI_Waitall until all messages are done

Musubi – Trace & insights – Improved communication pattern

- Processes run in lockstep
  - Synchronization from MPI_Allreduce
  - ⇒ Runtime differences accumulate
- No overlap of comm. and comp.
  - ~~MPI_Waitall until all messages are done~~
- Multiple patterns implemented
  - ⇒ Work on first messages

Musubi – Trace & insights – Improved communication pattern

- ~~Processes run in lockstep~~
  - ~~Synchronization from MPI_Allreduce~~
  - ⇒ ~~Runtime differences accumulate~~
- No overlap of comm. and comp.
  - ~~MPI_Waitall until all messages are done~~
- Convergence / error check
  - Delay by one timestep ⇒ No latency
  - Best case shown above
- Multiple patterns implemented
  - ⇒ Work on first messages

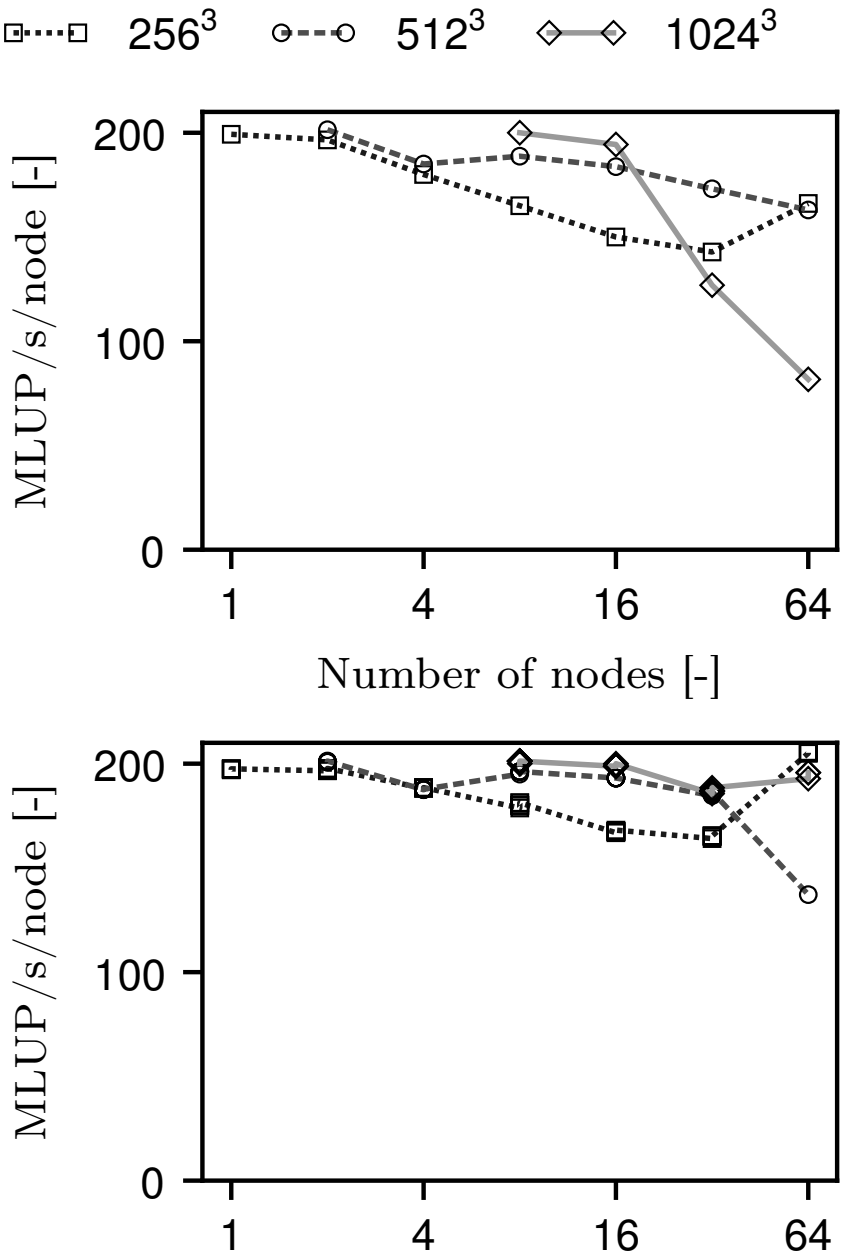
Musubi – Scalability study redone

Performance measurements

- Testcase ran on CARA
  - 64 processes per node
  - Increasing number of nodes
  - Increasing cells per node

Results

- Improved scalability
  - Even for low # of cells
- For 64 nodes very network-dependent



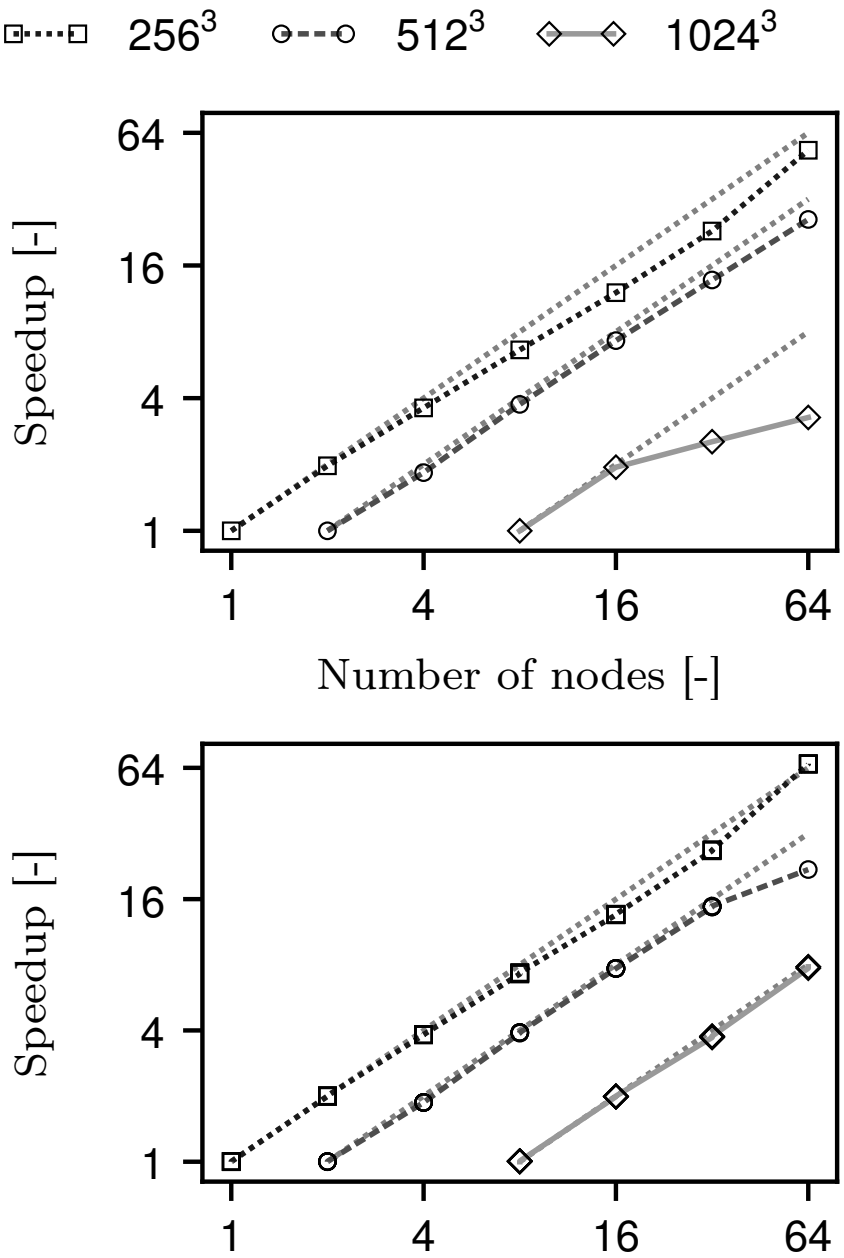
Musubi – Scalability study redone

Performance measurements

- Testcase ran on CARA
  - 64 processes per node
  - Increasing number of nodes
  - Increasing cells per node

Results

- Improved scalability
  - Even for low # of cells
- For 64 nodes very network-dependent



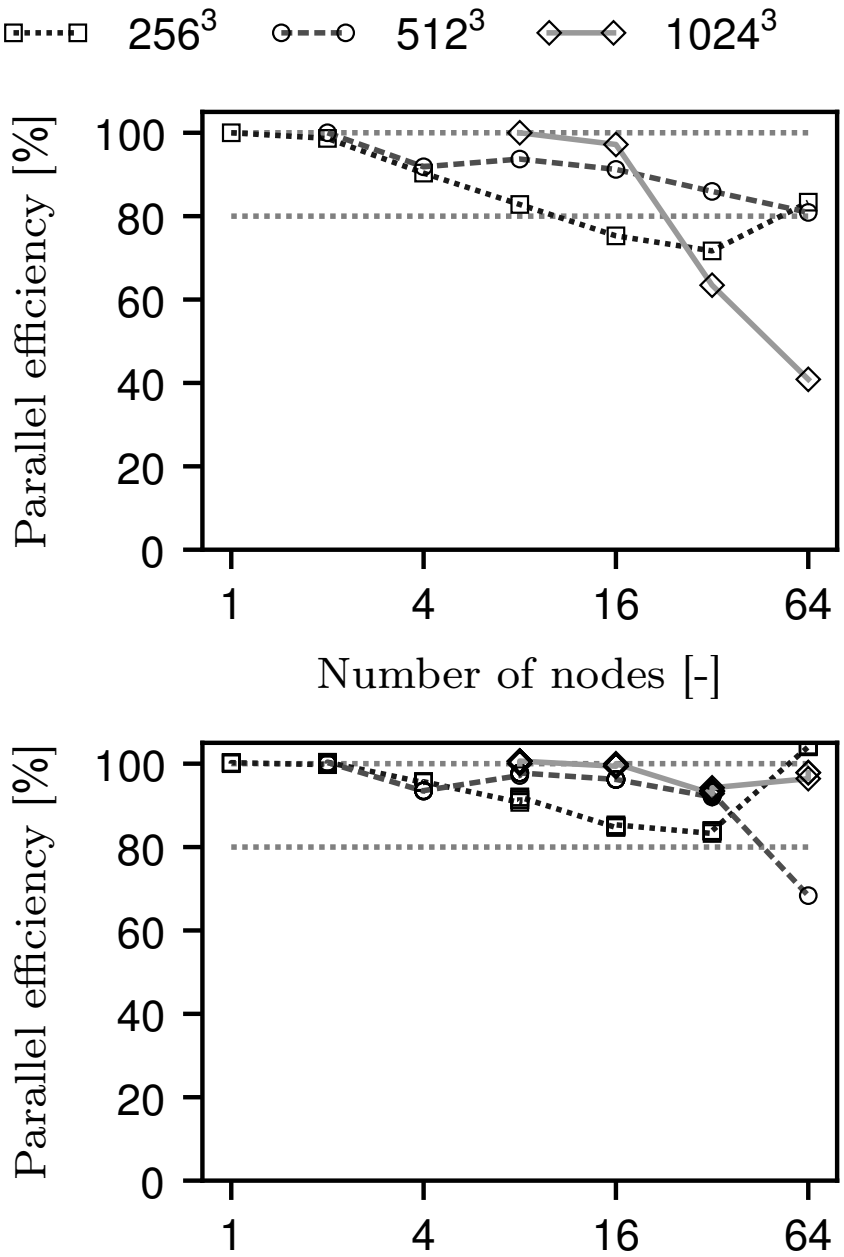
Musubi – Scalability study redone

Performance measurements

- Testcase ran on CARA
  - 64 processes per node
  - Increasing number of nodes
  - Increasing cells per node

Results

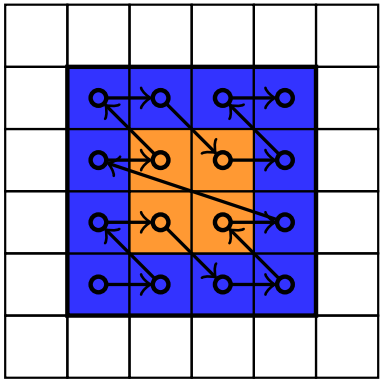
- Improved scalability
  - Even for low # of cells
- For 64 nodes very network-dependent
- Only overlap of comm. & comp. left

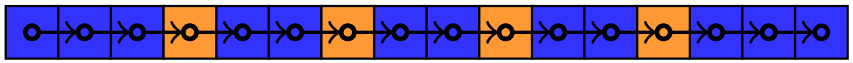


Musubi – Domain distribution and communication

Z-order curve

- Preserves data locality
- Mixes boundary and inner DoF
- Computation pattern
  - Work on all local DoF
  - Blocking send & receive data

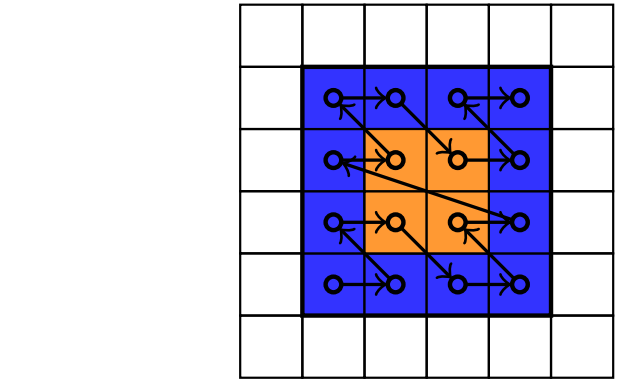


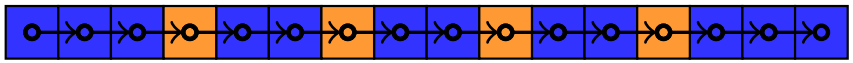


Musubi – Domain distribution and communication

Z-order curve

- Preserves data locality
- Mixes boundary and inner DoF
- Computation pattern
  - Work on all local DoF
  - Blocking send & receive data



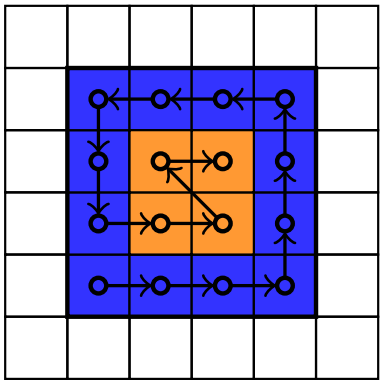


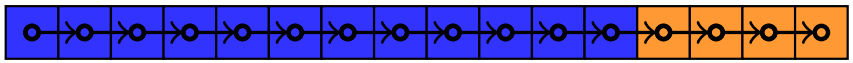
Boundary separation

- Separates boundary and inner DoF
- Computation pattern
  - Work on local boundary DoF
  - Send result & start receiving data
  - Work on inner DoF

⇒ Overlaps comm. & comp.

  - Lower cache efficiency





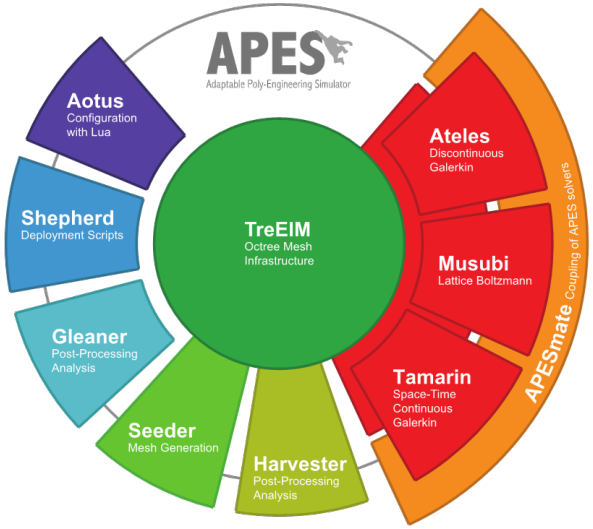
Musubi – Summary

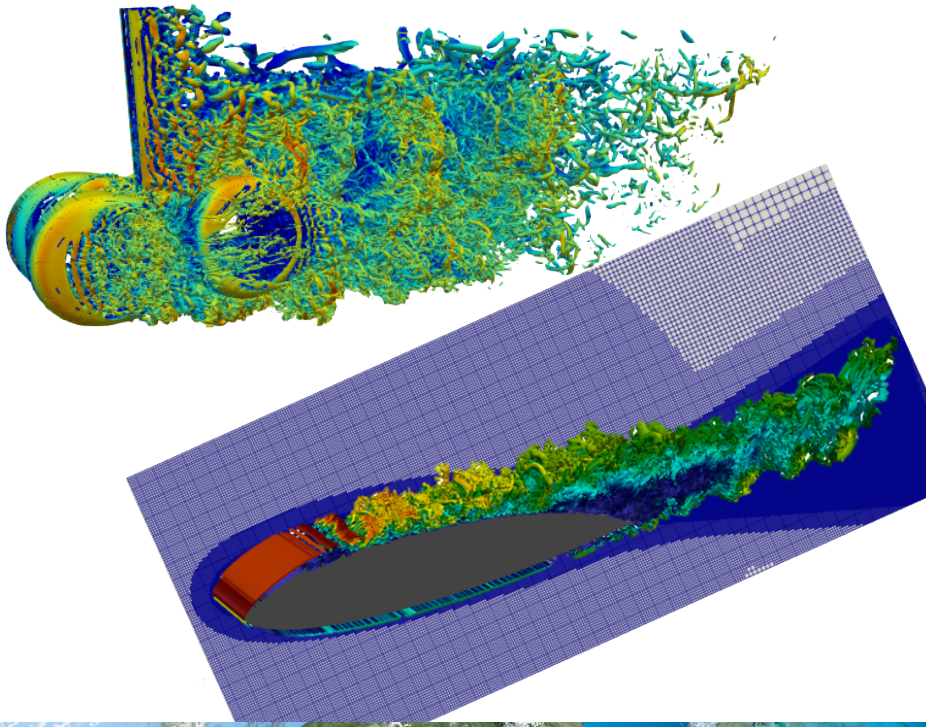
Summary

- Performance evaluation of Musubi
- Overall good scalability
- Degradation at high core counts
  - Processes in lockstep
  - Blocking `MPI_Allreduce()`
  - No overlap of comp. & comm. yet

Next steps & future work

- Investigate boundary separation
- Evaluation for multi-level grids
- Investigate AuxField handling

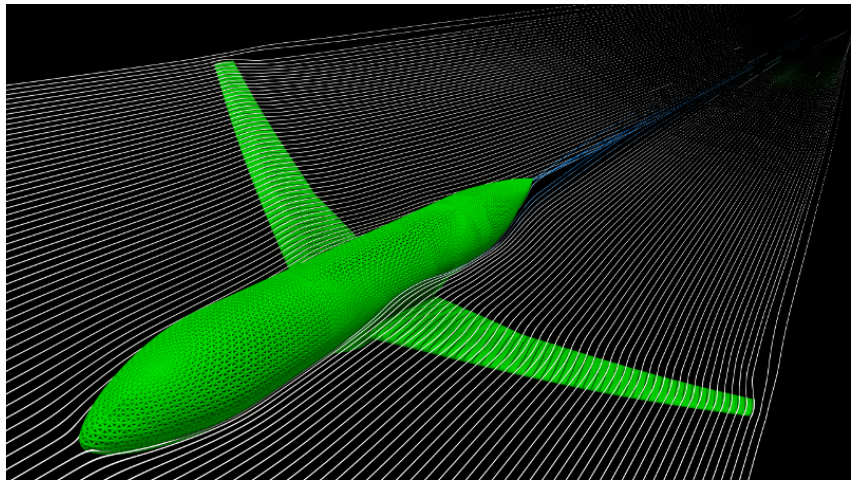




Performance analysis of CODA

CODA

- Flow solver written by
  - ONERA
  - DLR
  - Airbus
  - Many other European partners



Coding

- User scenarios written in Python
- Solver core written in C++
- Hybrid parallelization
  - Coarse-grain domain decomposition
    - MPI / Gaspi
  - Fine-grain domain decomposition
    - Thread parallelism
- Computation & communication overlap
- Spliss as linear solver

CODA – Testcase

Common Research Model (CRM)

Drag Prediction Workshop #5

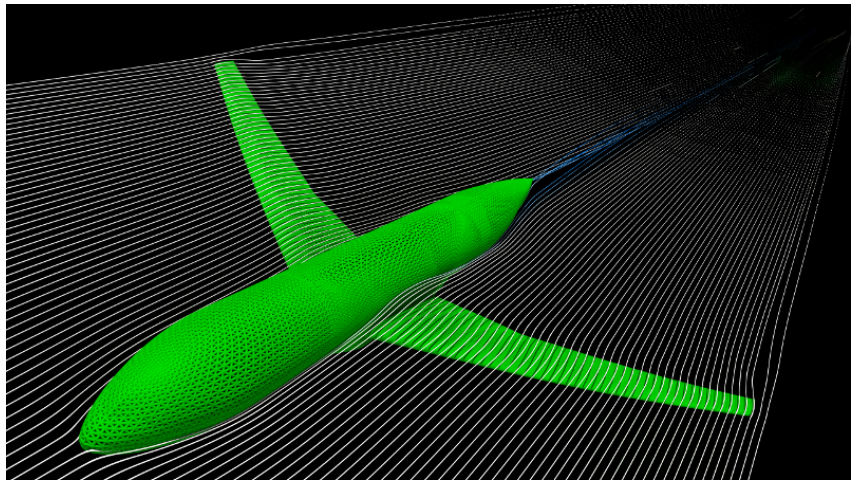
- RANS SA negative
- Mach number $Ma = 0.85$
- Angle of attack $\alpha = 2.209$ degrees

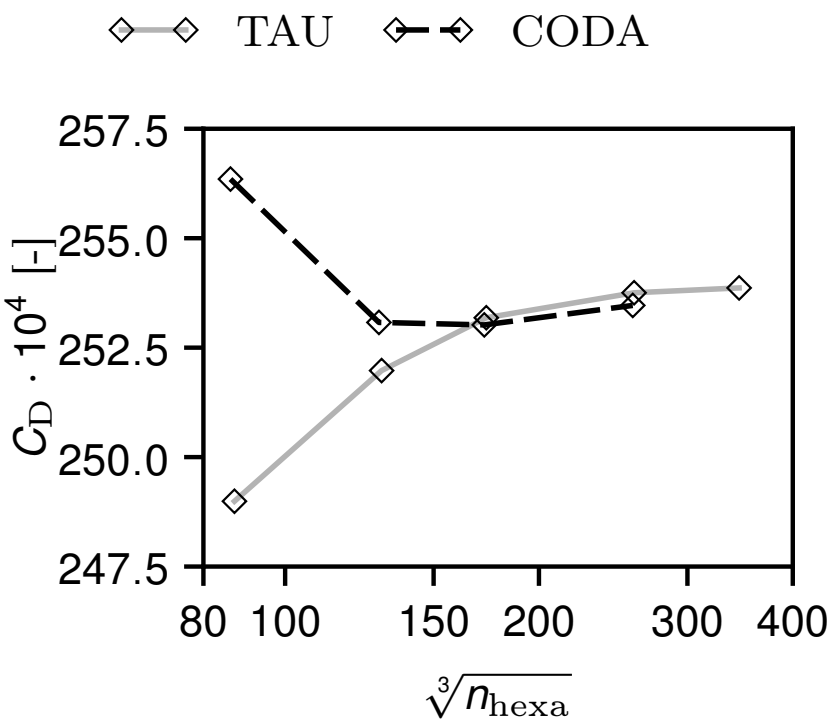
Solver configuration

- Linearized implicit Euler solver
- GMRES(20) + Jacobi(50) + LU
- FD for linear operator

Linearized implicit Euler time step

1. Compute time step size
2. Compute residual
3. Compute Jacobian
4. Solve linear system
5. Apply solution





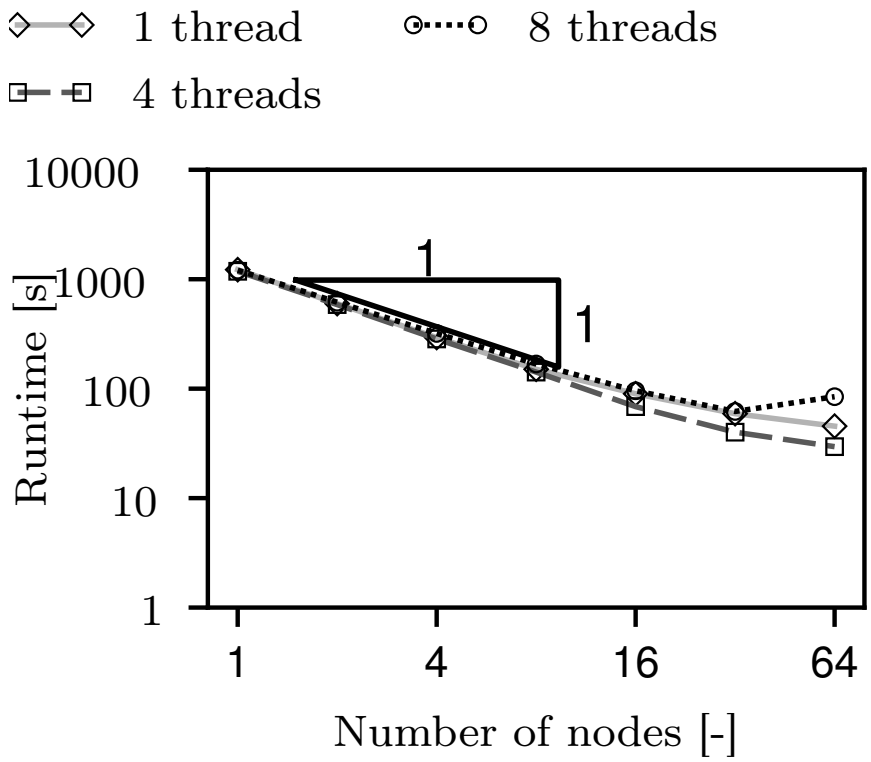
CODA – Scalability study

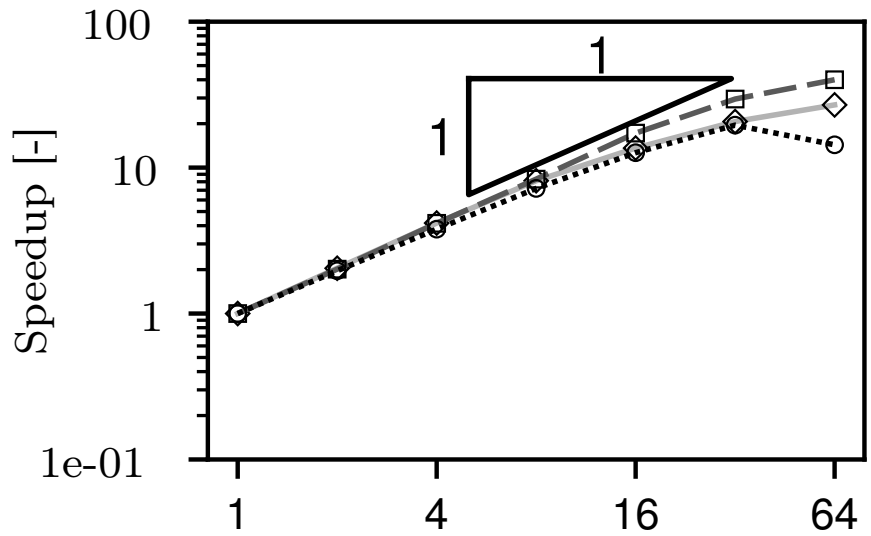
CRM

- Structured “tiny” grid, split into prisms
  - $1.3 \cdot 10^6$ prism elements
- Run on CARA, 64 cores per node
- Nodes $\in \{1, \ldots, 64\}$

1 node: $\approx$ 20000 elements per core

64 nodes: $\approx$ 320 elements per core





CODA – Scalability study

CRM

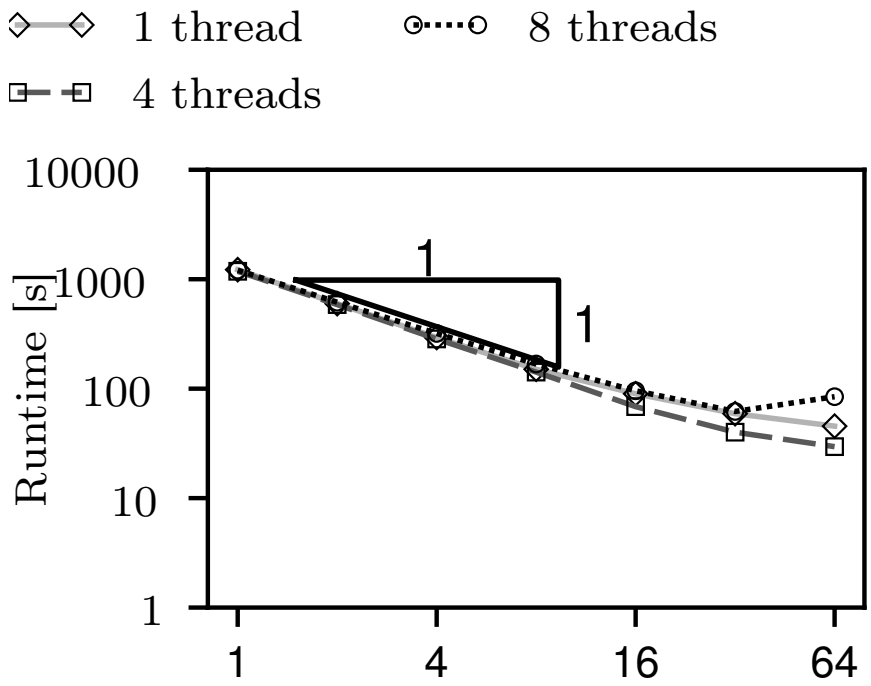
- Structured “tiny” grid, split into prisms
  - $1.3 \cdot 10^6$ prism elements
- Run on CARA, 64 cores per node
- Nodes $\in \{1, \ldots, 64\}$

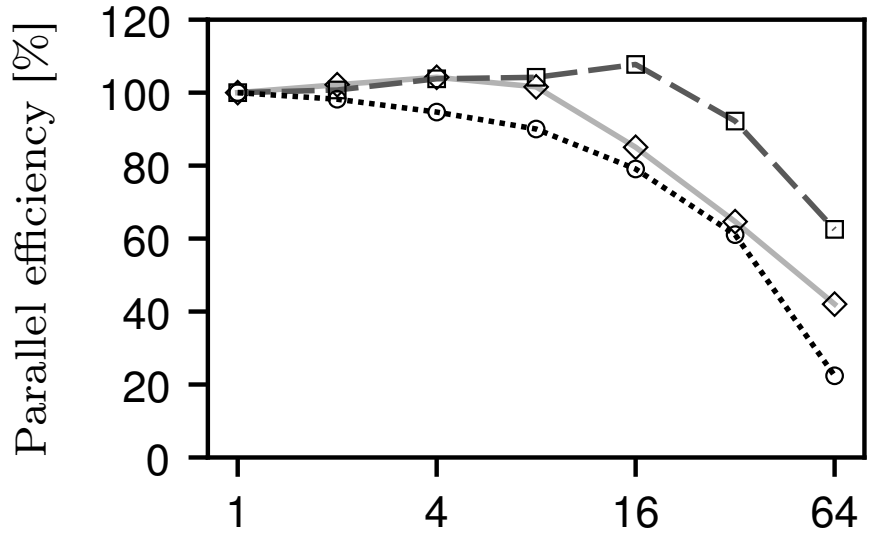
1 node: $\approx$ 20000 elements per core

64 nodes: $\approx$ 320 elements per core

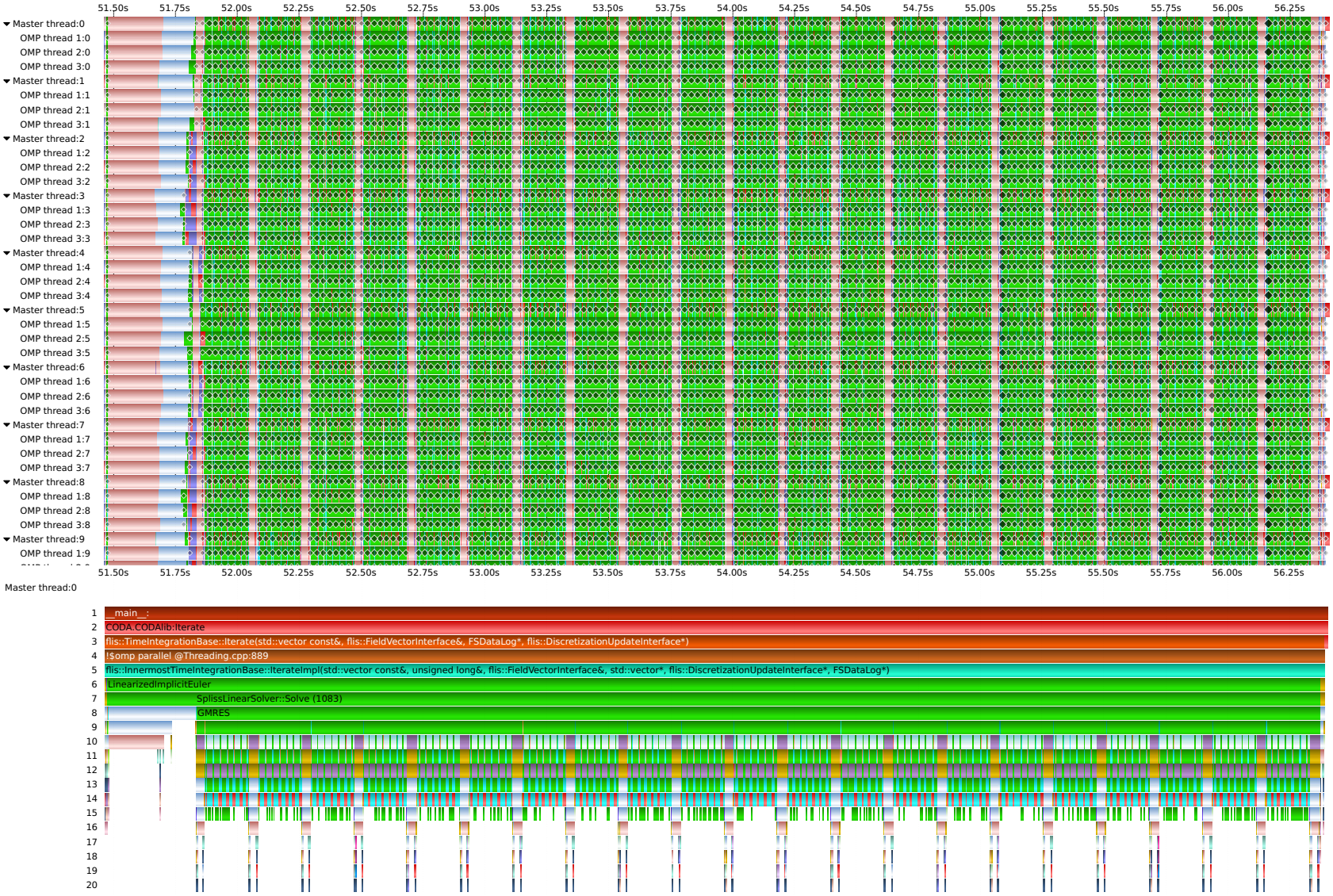
Insights

- Using hybrid parallelization imperative
  - Solver: 90 % efficiency on 32 nodes
- CPU architecture Zen
  - L3 cache on 4 core chiplet
  - Comm. on chiplet via L3 cache
  - Comm. between chiplets over memory
  - $\Rightarrow$ 4 threads perform better than 8

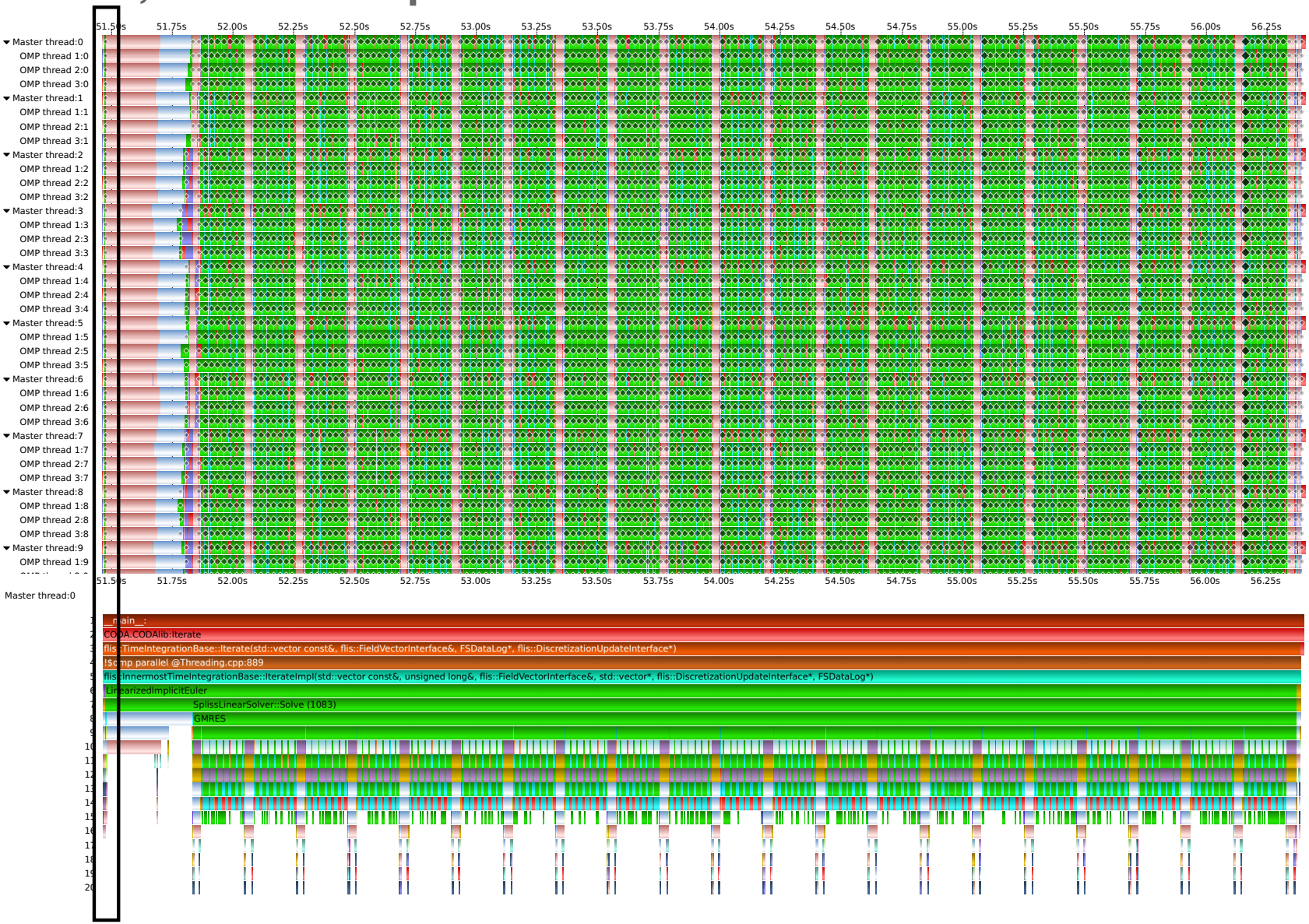




CODA – Trace, last timestep

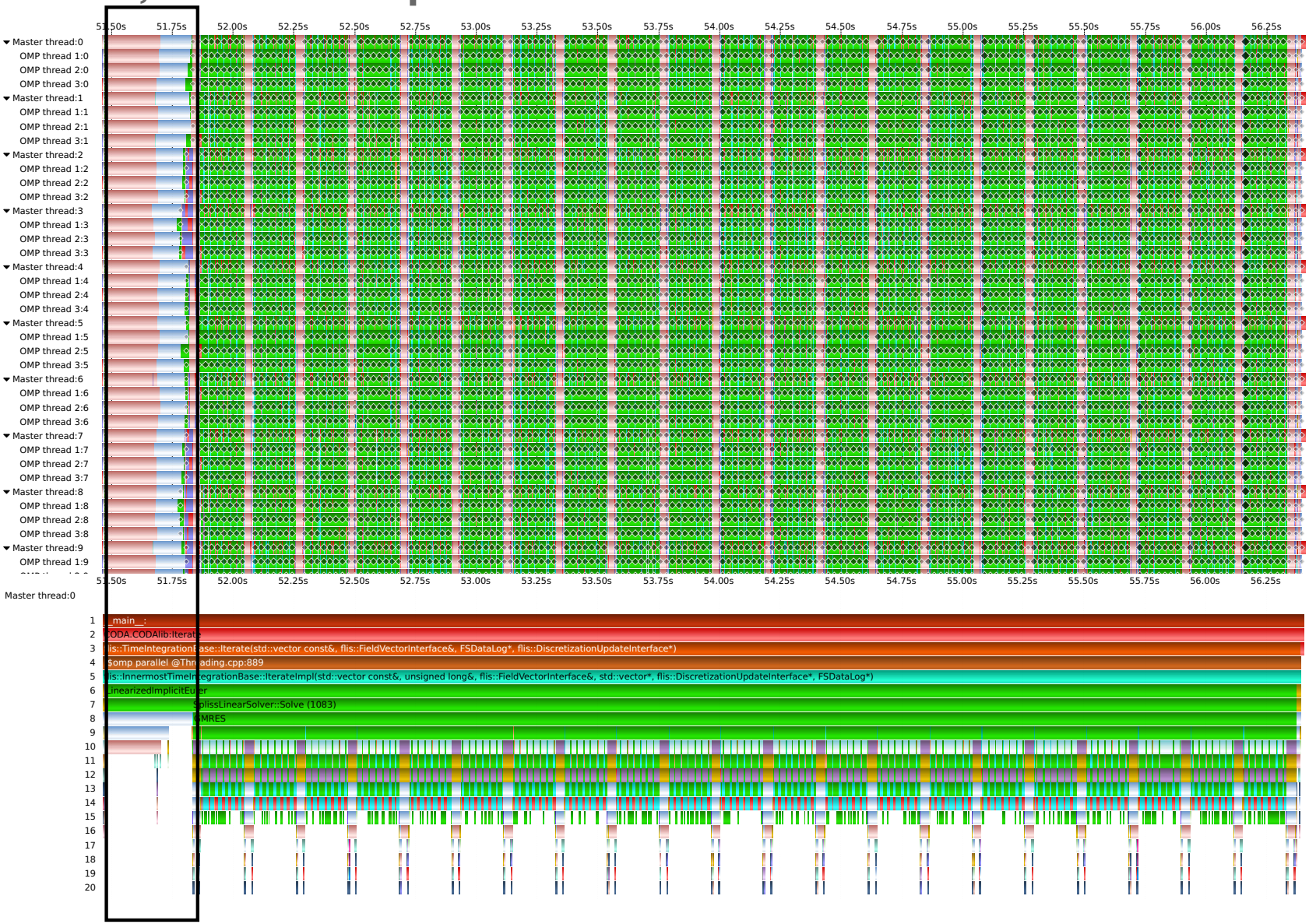


CODA – Trace, last timestep



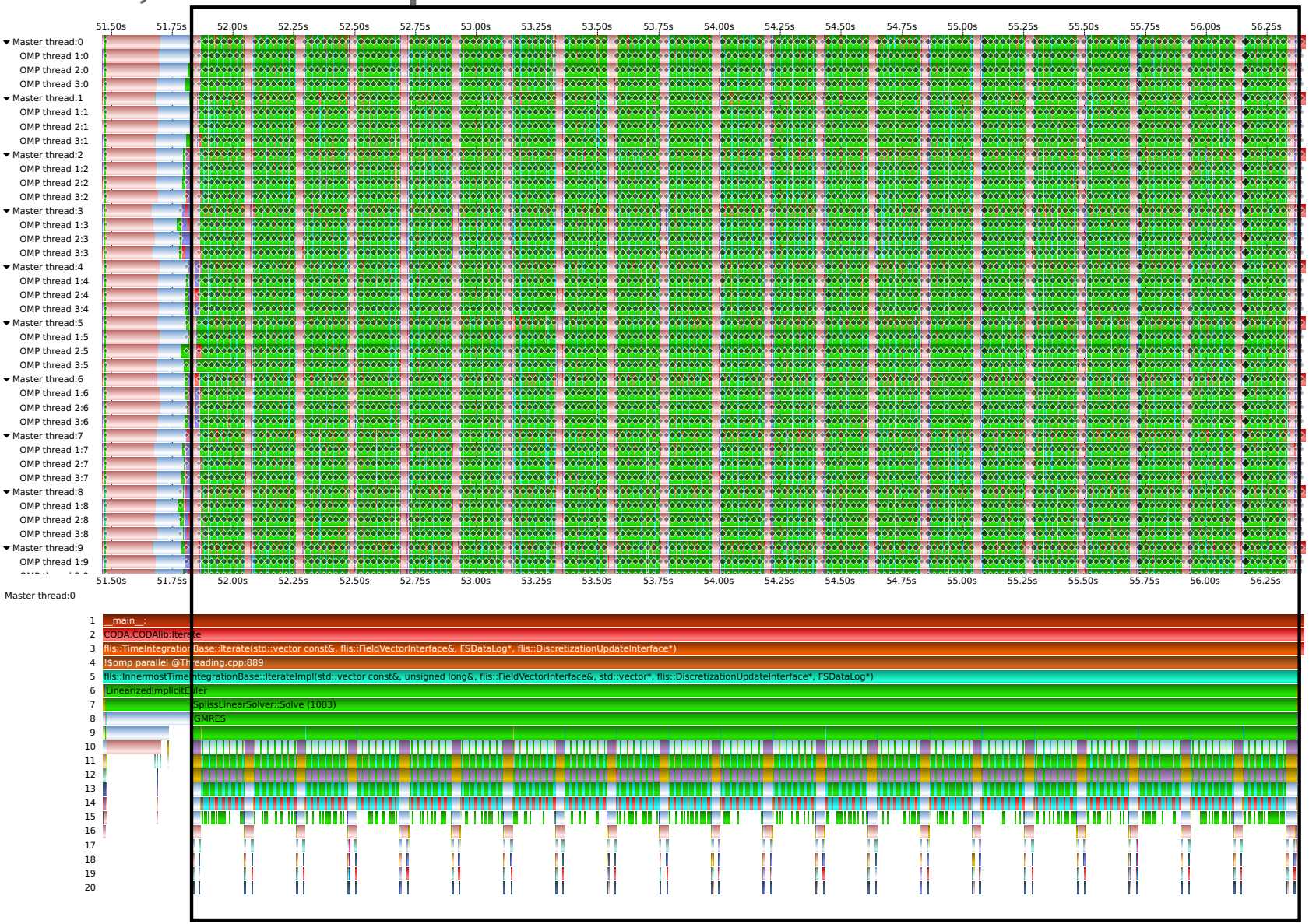
Residual evaluation

CODA – Trace, last timestep



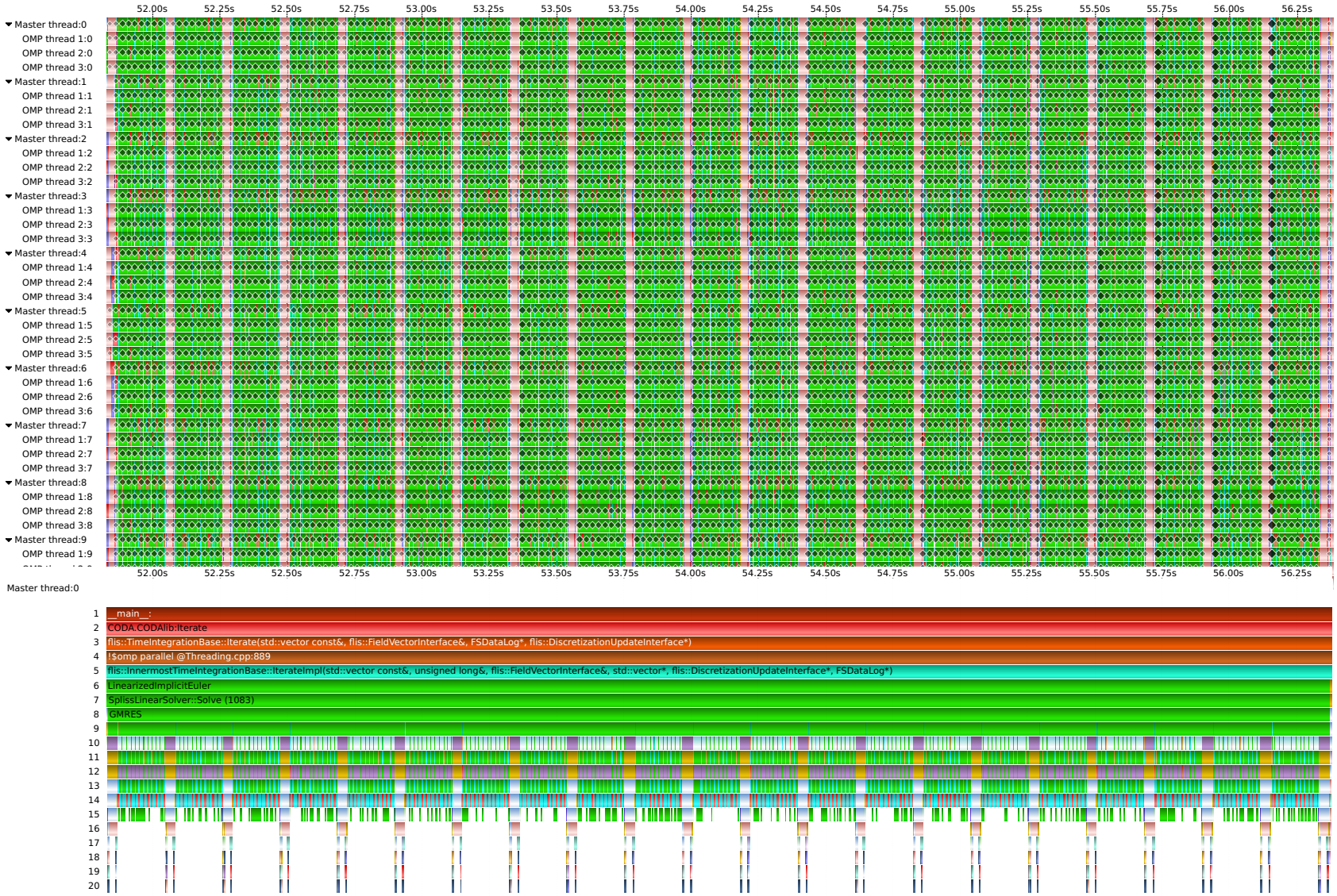
Jacobian computation ≈ 7 %

CODA – Trace, last timestep

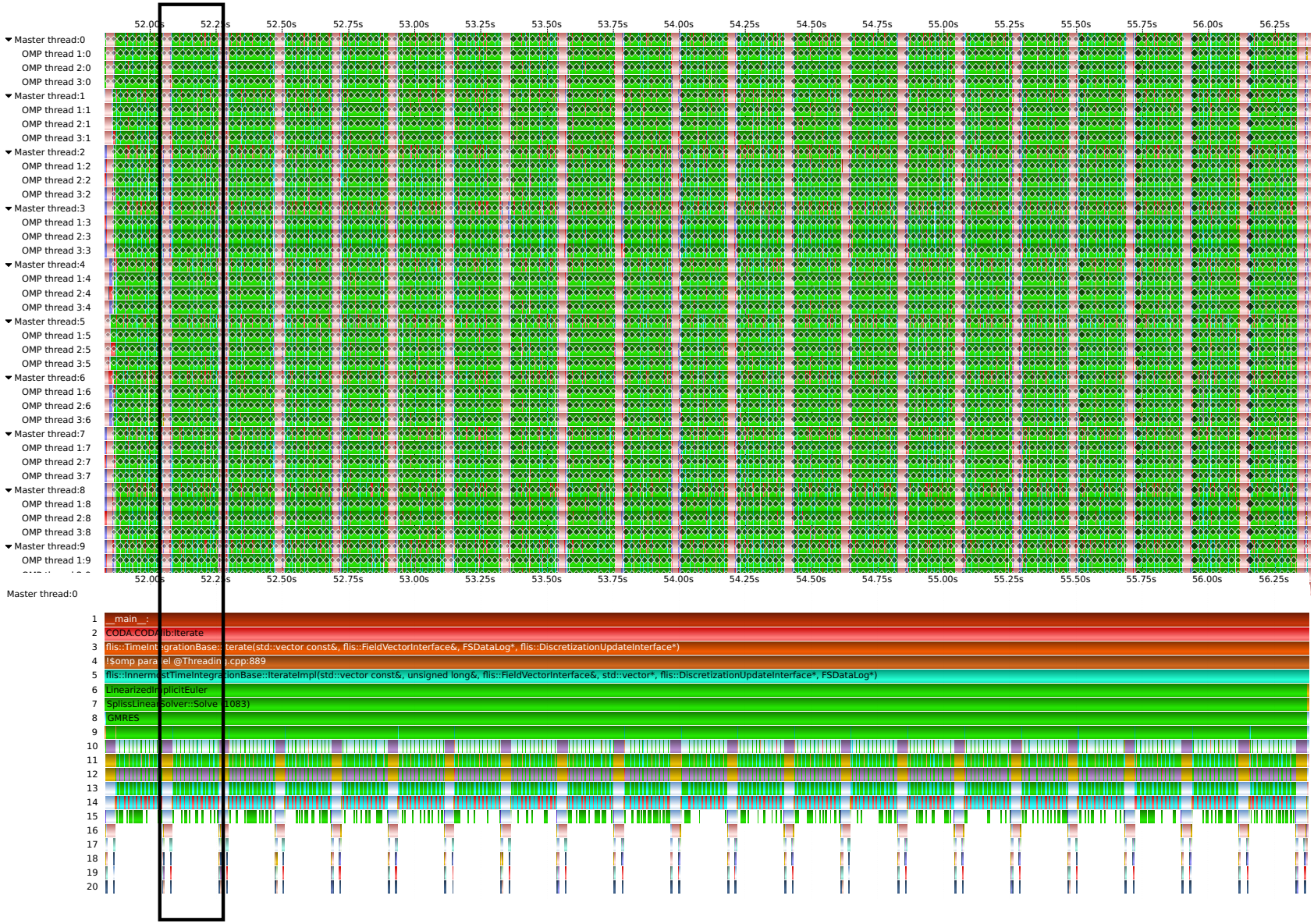


GMRES solver ≈ 92% ⇒ focus on Spliss

CODA – Trace, GMRES linear solver

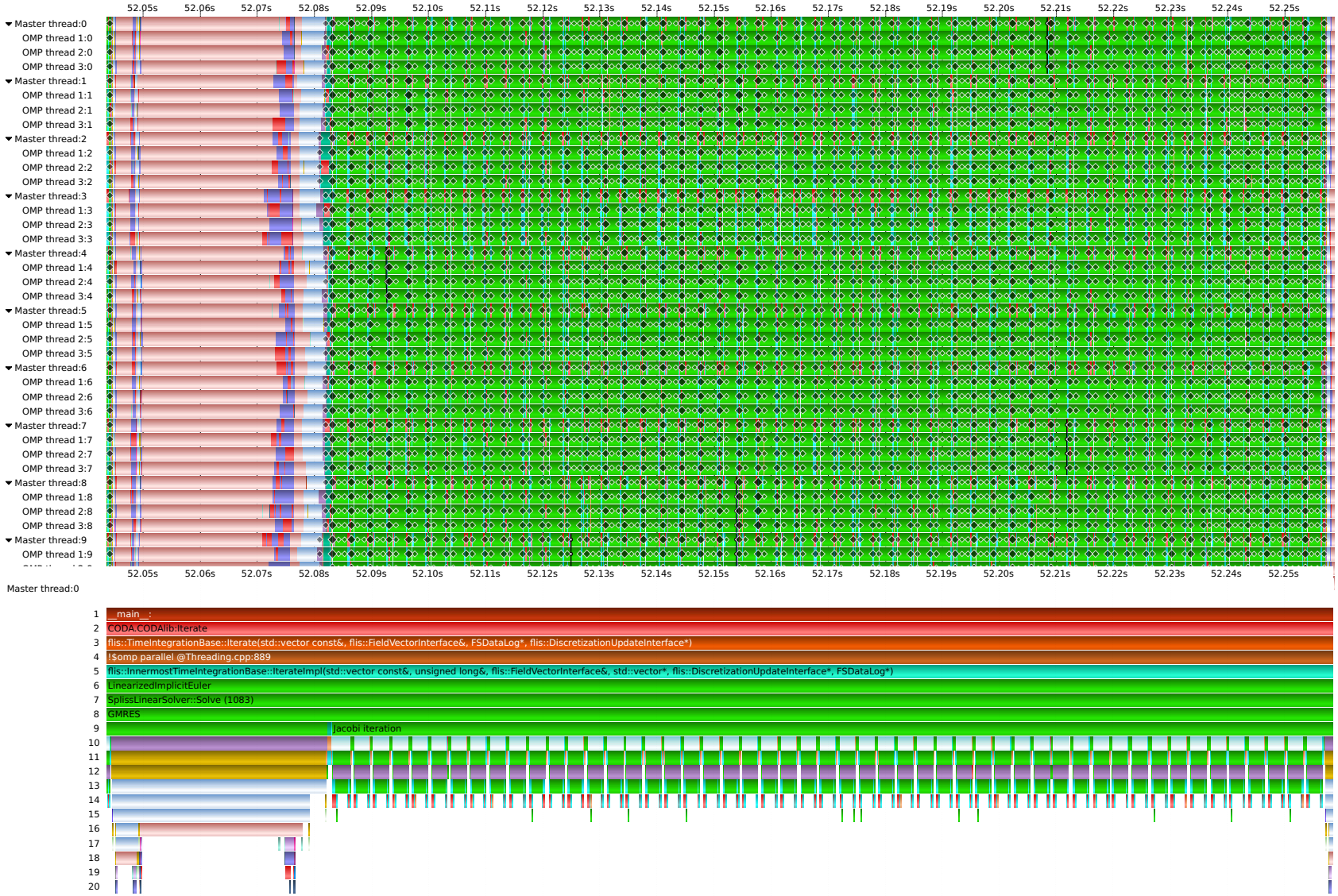


CODA – Trace, GMRES linear solver



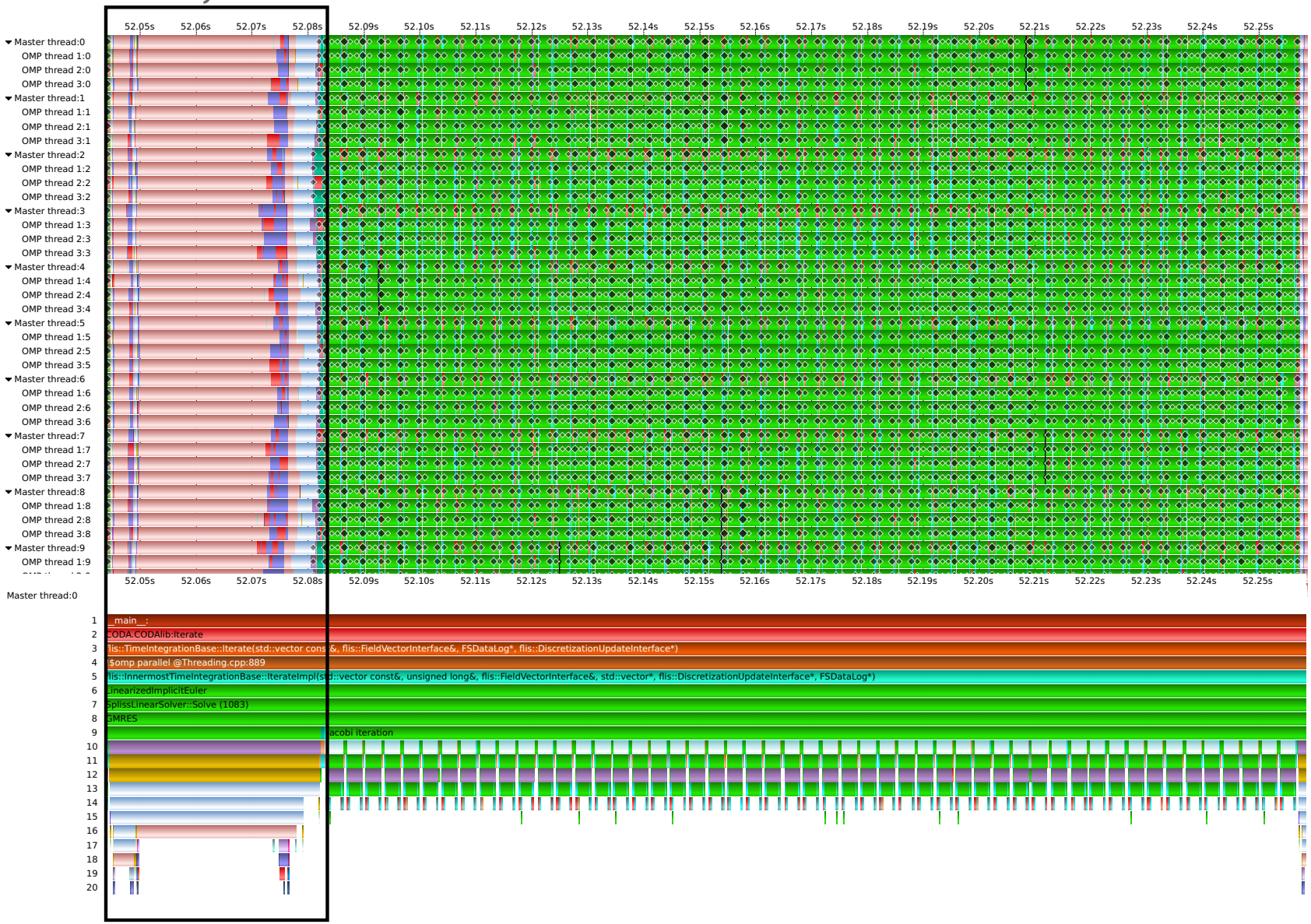
One iteration

CODA – Trace, one GMRES iteration



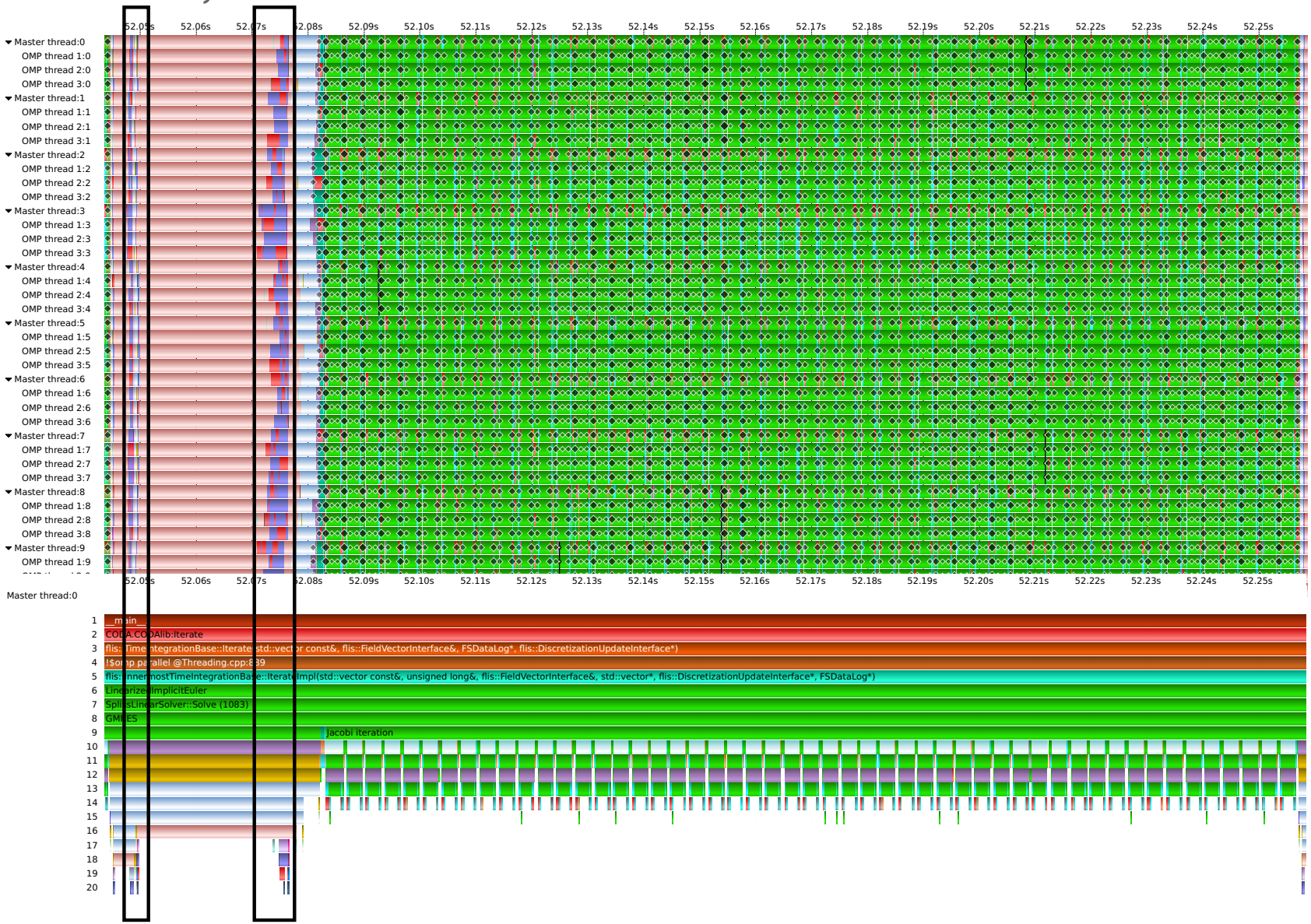
DLR

CODA – Trace, one GMRES iteration



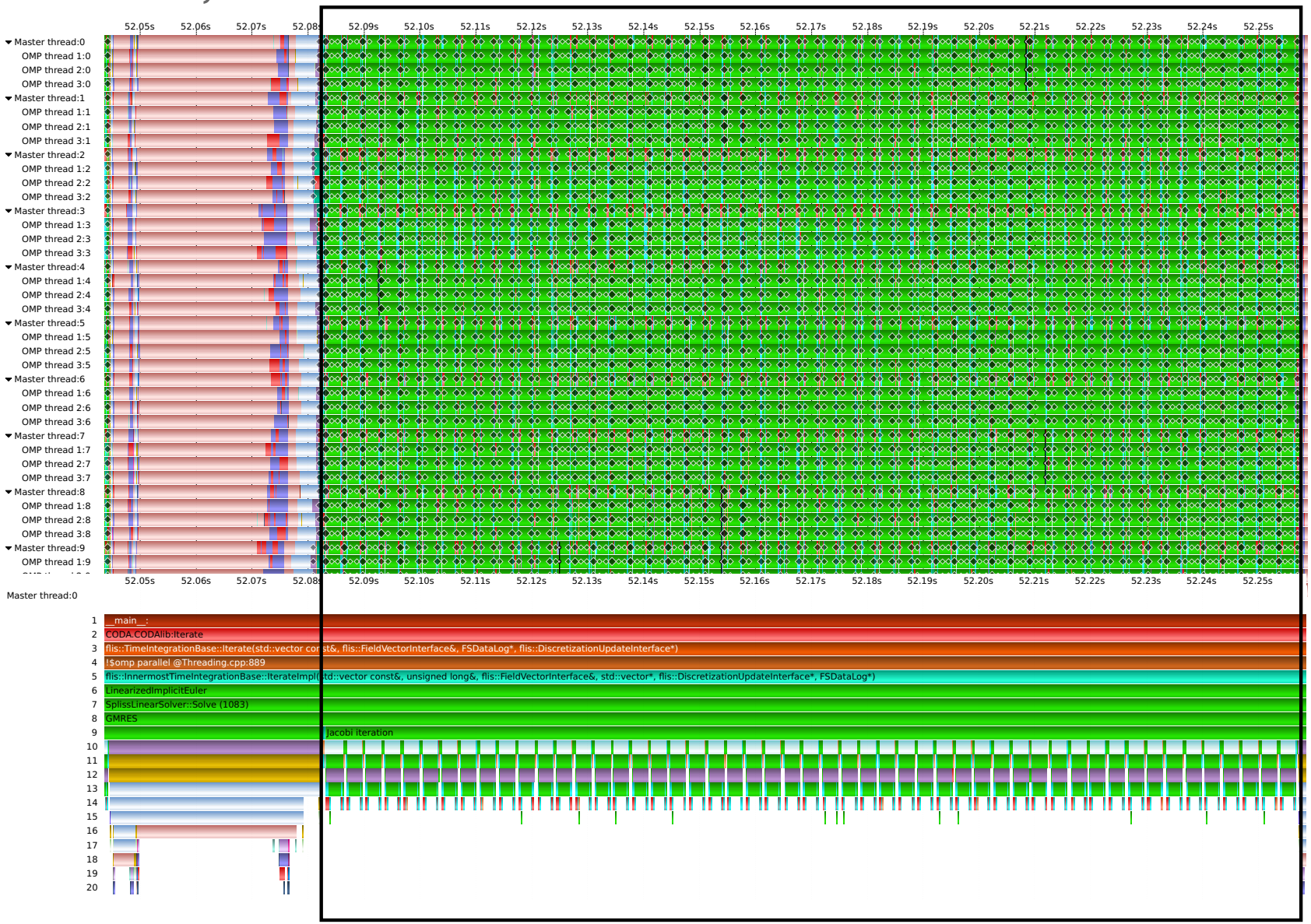
AD matrix-vector product

CODA – Trace, one GMRES iteration



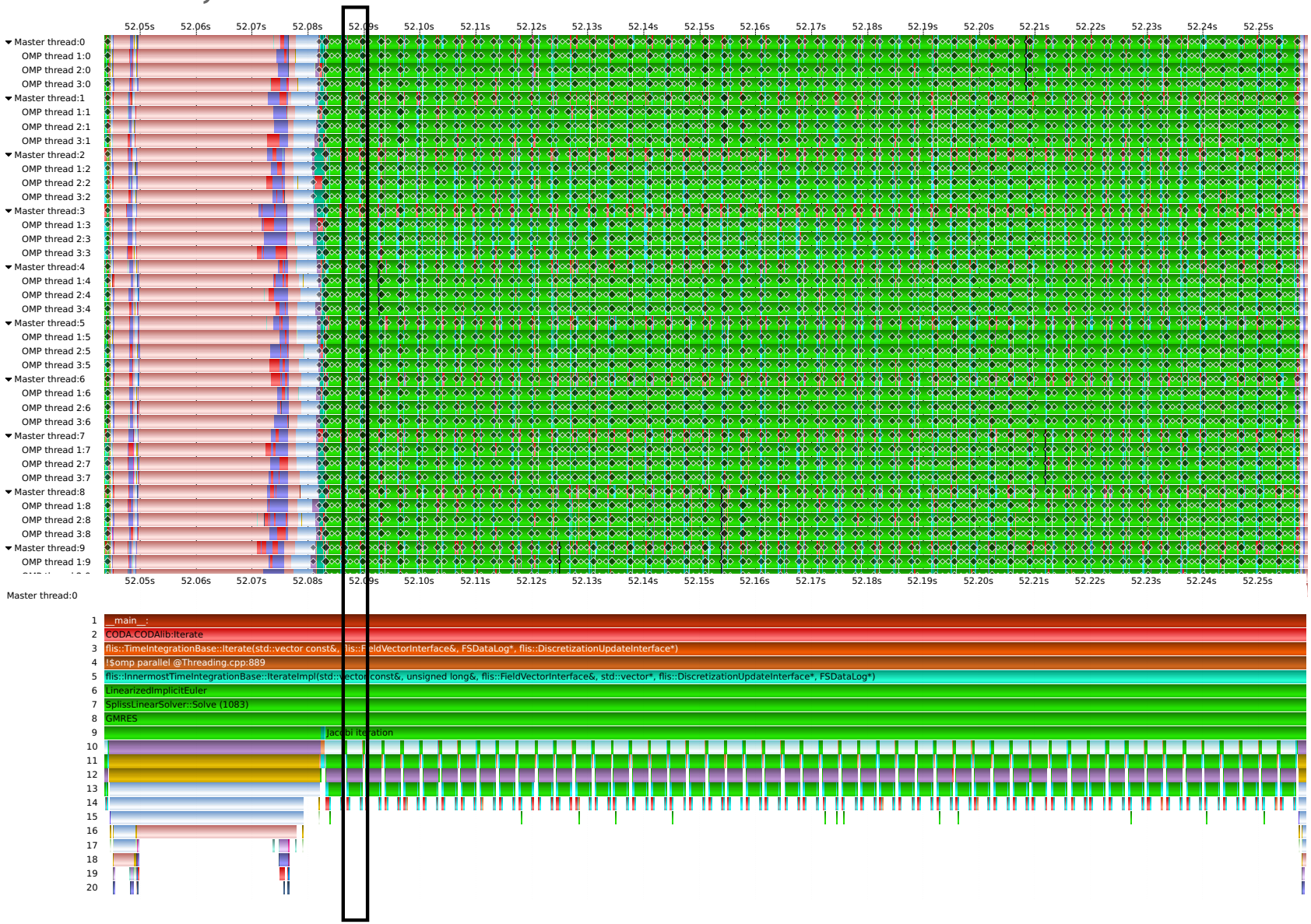
communication

CODA – Trace, one GMRES iteration



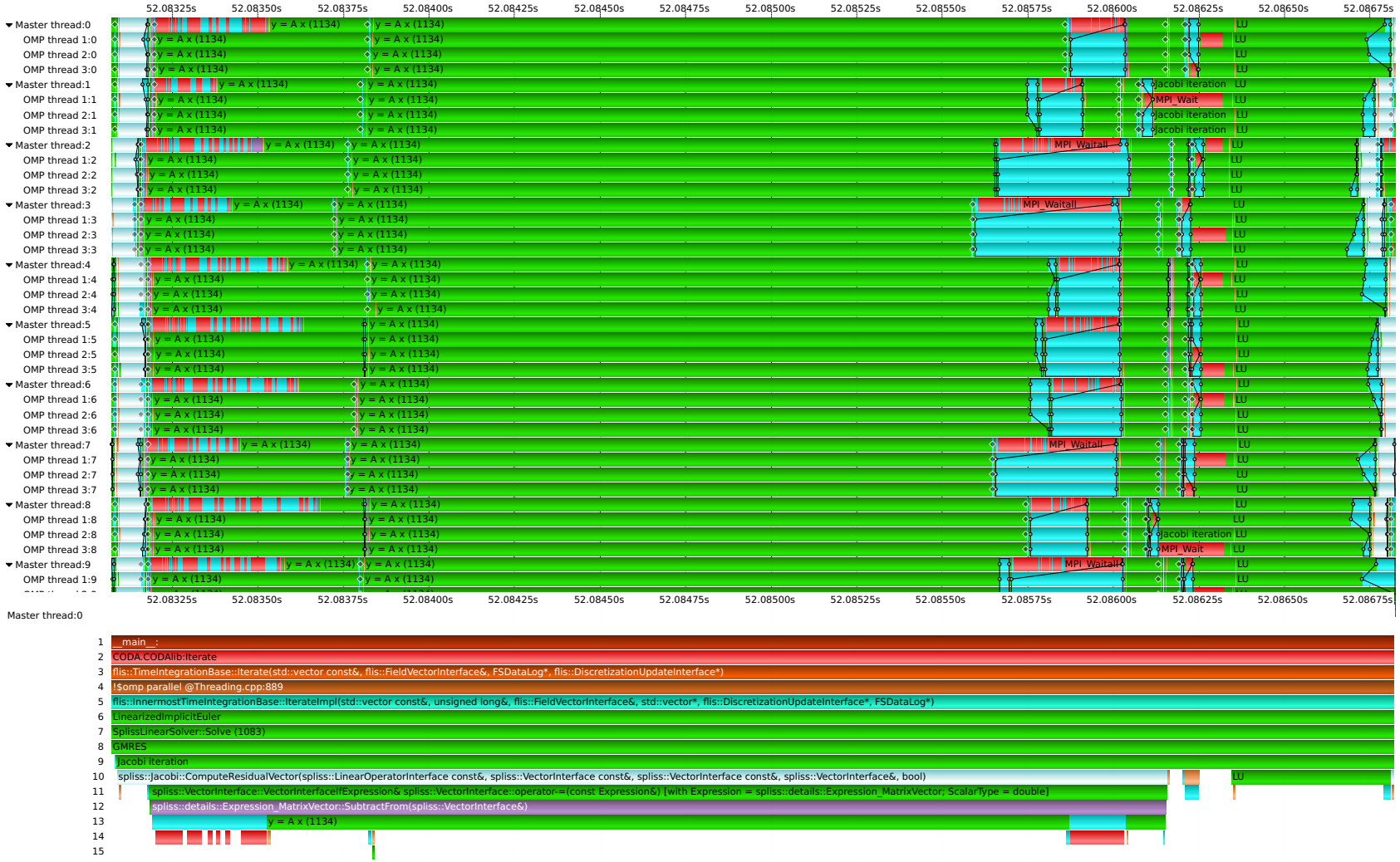
Jacobi preconditioner

CODA – Trace, one GMRES iteration

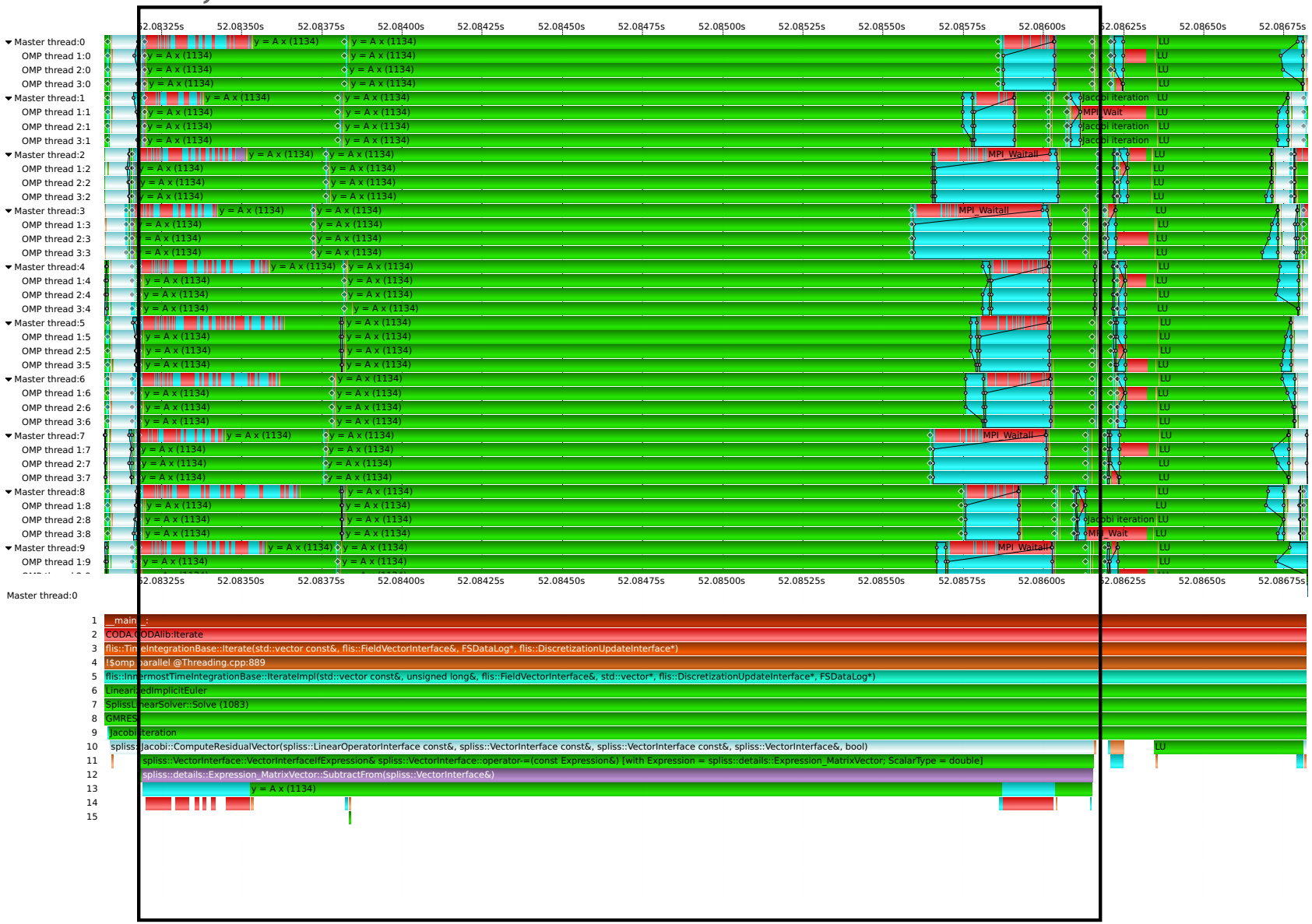


One Jacobi iteration

CODA – Trace, one Jacobi iteration

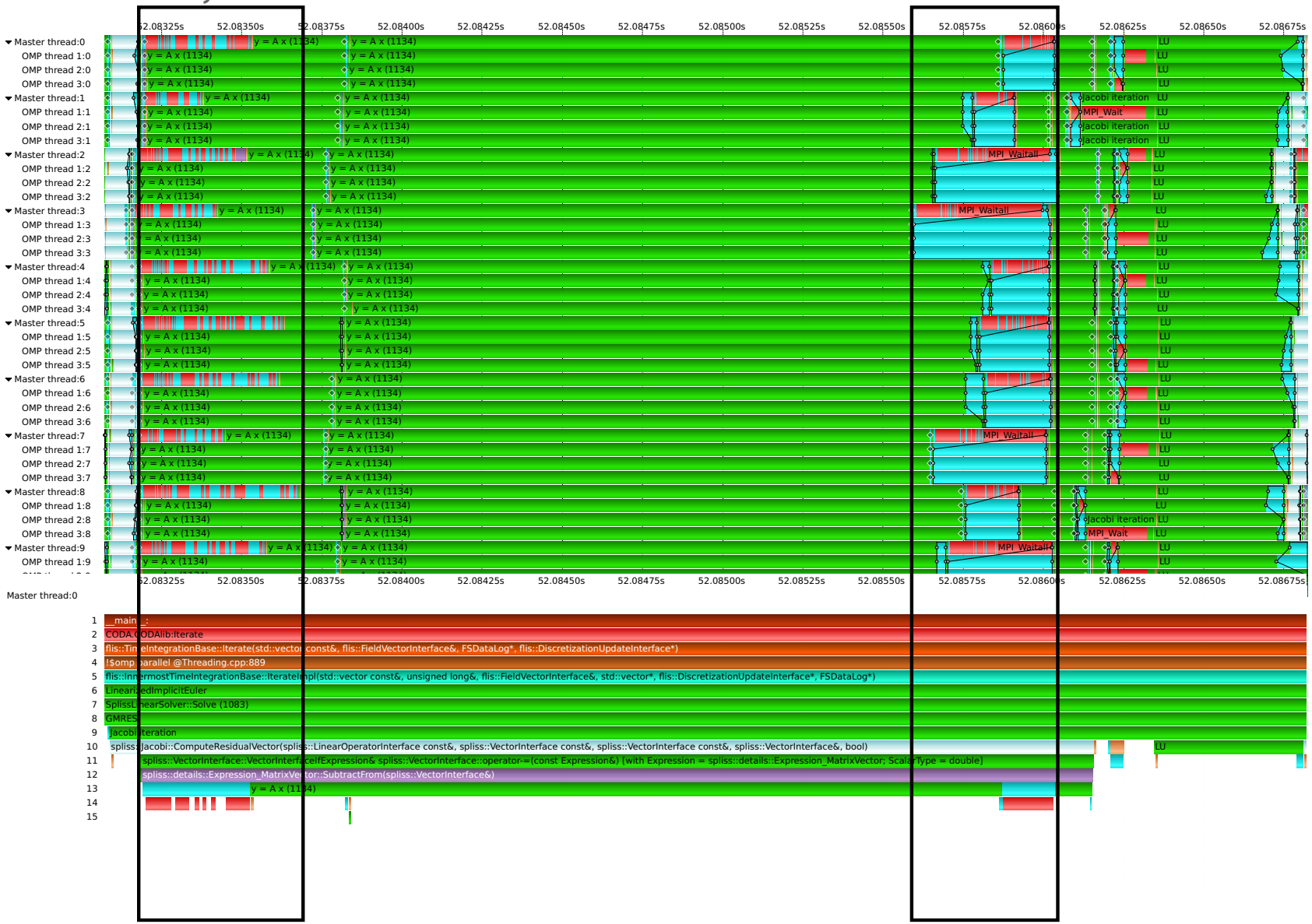


CODA – Trace, one Jacobi iteration



Matrix-vector product

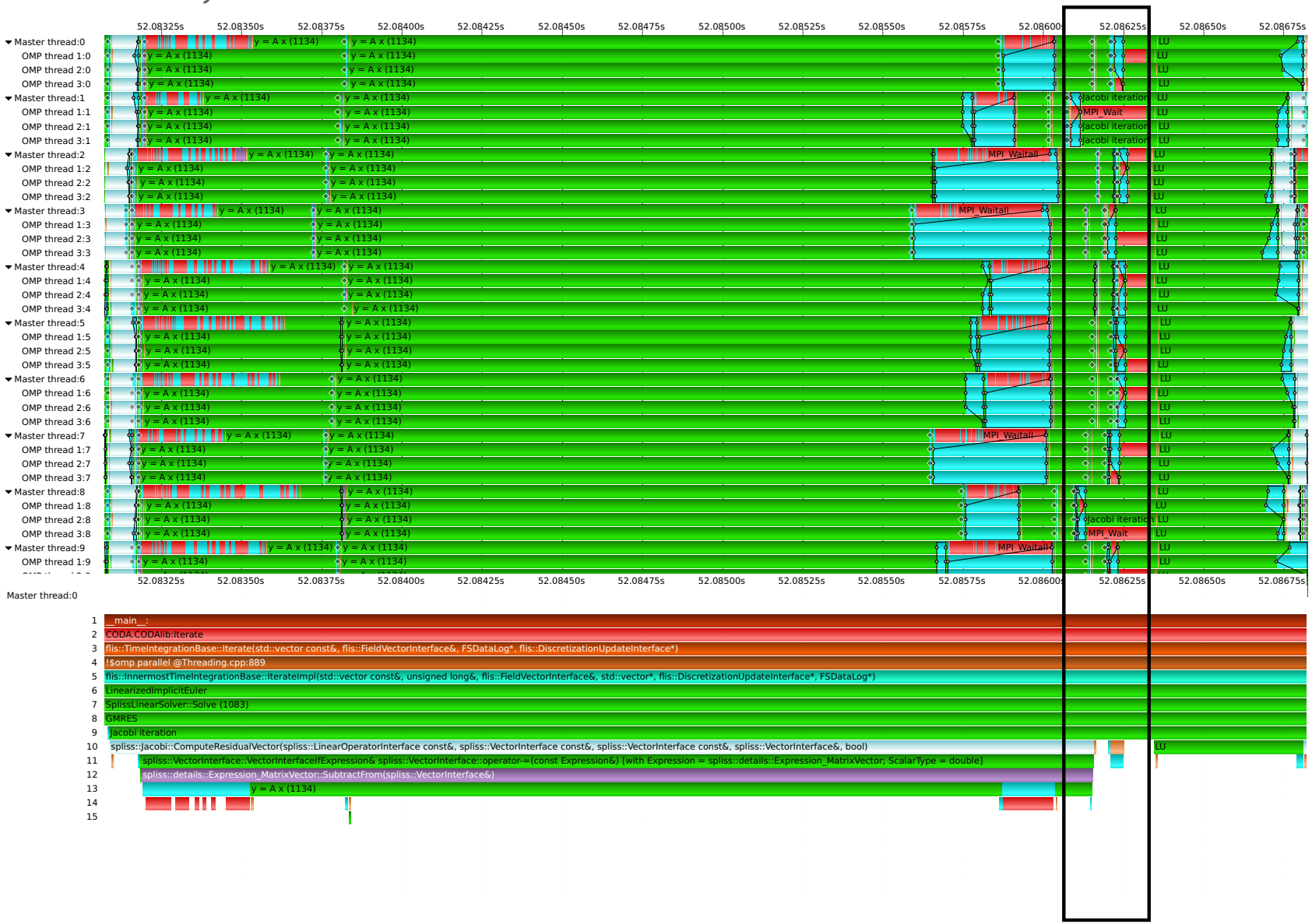
CODA – Trace, one Jacobi iteration



Start halo exchange

End halo exchange

CODA – Trace, one Jacobi iteration



Stray IAllreduce & Wait

CODA – Summary

CODA

- Very good scalability
  - Automatic overlap of comm. & comp.
  - Few *MPI_Allreduce*
- Largest portion spent in Spliss
  - ⇒ Concentrate on Spliss side

Spliss

- Good overlap of comm. & comp.
  - But for CODA linear operators
  - AD matrix-vector product
  - Full Jacobian resolves problem
  - Similar problems expected for DGSEM
- `$omp barrier` after every kernel
  - Potential for less synchronization
- Stray IAllreduce due to CODA wrapping
- No global coupling

CODA – Summary

CODA

- Very good scalability
  - Automatic overlap of comm. & comp.
  - Few *MPI_Allreduce*
- Largest portion spent in Spliss
  - ⇒ Concentrate on Spliss side

Spliss

Spliss

- Good overlap of comm. & comp.
  - But for CODA linear operators
  - AD matrix-vector product
  - Full Jacobian resolves problem
  - Similar problems expected for DGSEM
- `$omp barrier` after every kernel
  - Potential for less synchronization
- Stray IAllreduce due to CODA wrapping
- No global coupling

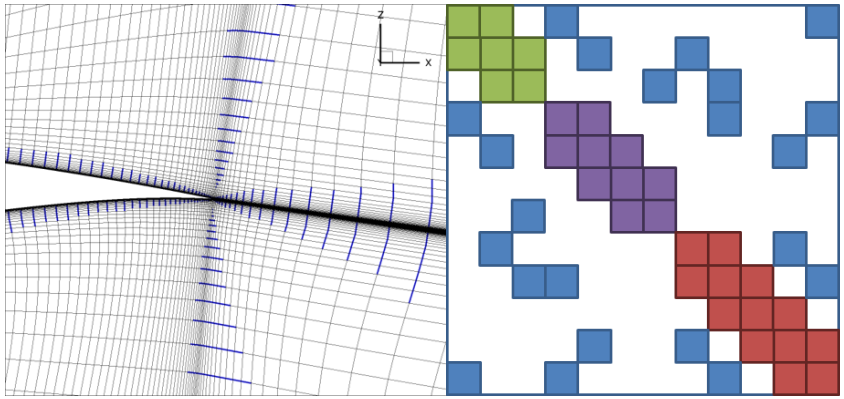
Next steps: Spliss

- Currently no global coupling
  - # of iterations increases with grid size
  - Lines inversion
  - Multigrid
- Largest runtime portion in prec.
  - Does not need to be exact
  - ⇒ Mixed precision
  - First implementation
  - ⇒ Up to ≈ 30 % performance gain

CODA – Summary

CODA

- Very good scalability
  - Automatic overlap of comm. & comp.
  - Few *MPI_Allreduce*
- Largest portion spent in Spliss
  - ⇒ Concentrate on Spliss side



Spliss

- Good overlap of comm. & comp.
  - But for CODA linear operators
  - AD matrix-vector product
  - Full Jacobian resolves problem
  - Similar problems expected for DGSEM
- `$omp barrier` after every kernel
  - Potential for less synchronization
- Stray IAllreduce due to CODA wrapping
- No global coupling

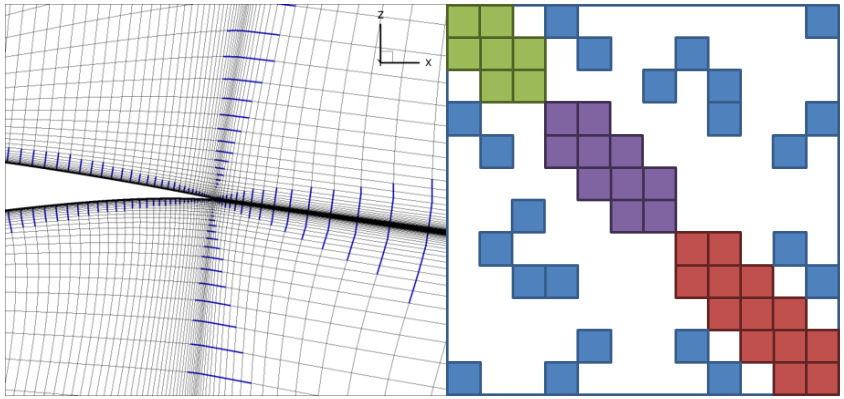
Next steps: Spliss

- Currently no global coupling
  - # of iterations increases with grid size
  - **Lines inversion**
  - Multigrid
- Largest runtime portion in prec.
  - Does not need to be exact
  - ⇒ Mixed precision
  - First implementation
  - ⇒ Up to ≈ 30 % performance gain

CODA – Summary

CODA

- Very good scalability
  - Automatic overlap of comm. & comp.
  - Few *MPI_Allreduce*
- Largest portion spent in Spliss
  - ⇒ Concentrate on Spliss side



Spliss

- Good overlap of comm. & comp.
  - But for CODA linear operators
  - AD matrix-vector product
  - Full Jacobian resolves problem
  - Similar problems expected for DGSEM
- `$omp barrier` after every kernel
  - Potential for less synchronization
- Stray IAllreduce due to CODA wrapping
- No global coupling

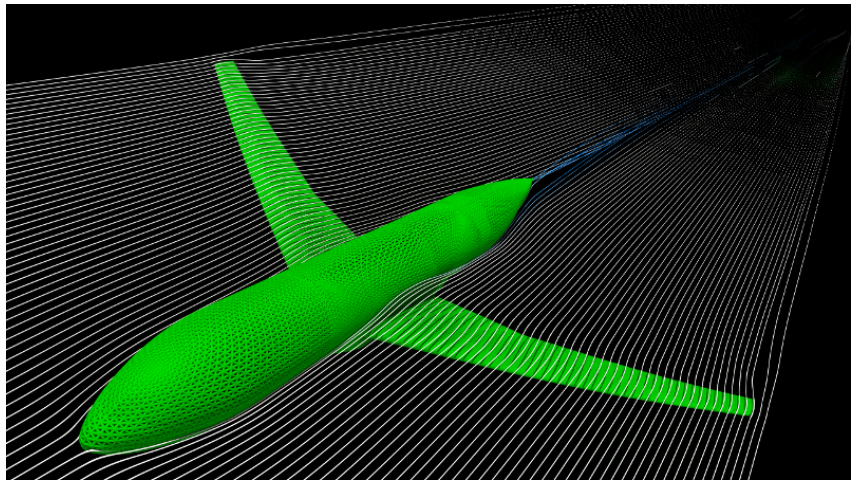
Next steps: Spliss

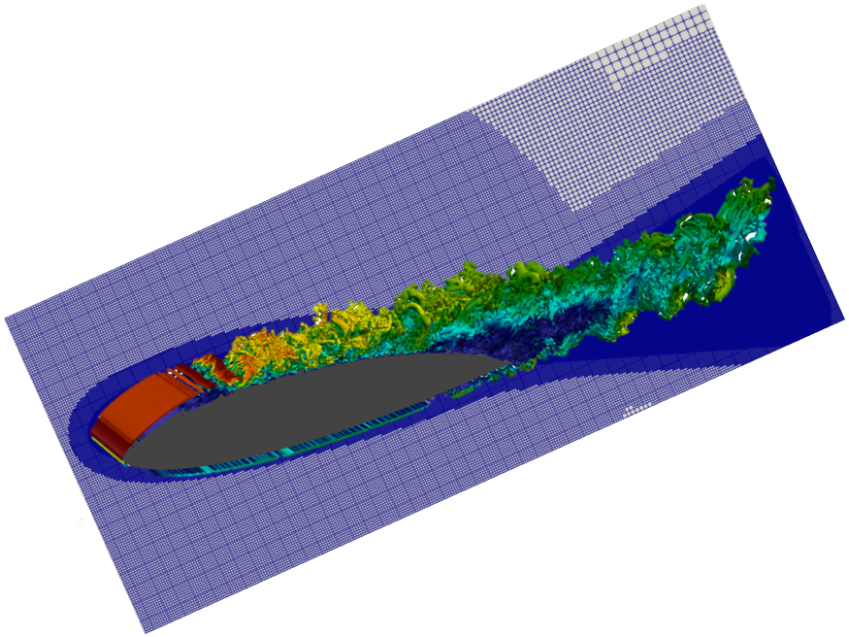
- Currently no global coupling
  - # of iterations increases with grid size
  - Lines inversion
  - Multigrid
- Largest runtime portion in prec.
  - Does not need to be exact
  - ⇒ Mixed precision
  - First implementation
  - ⇒ Up to ≈ 30 % performance gain
- **Benefits directly transfer to TRACE**

Summary & outlook

Summary

- Case studies shown
  - Scalability studies
  - Tracing of application
  - Performance bottlenecks identified
  - Solutions proposed
  - First gains demonstrated

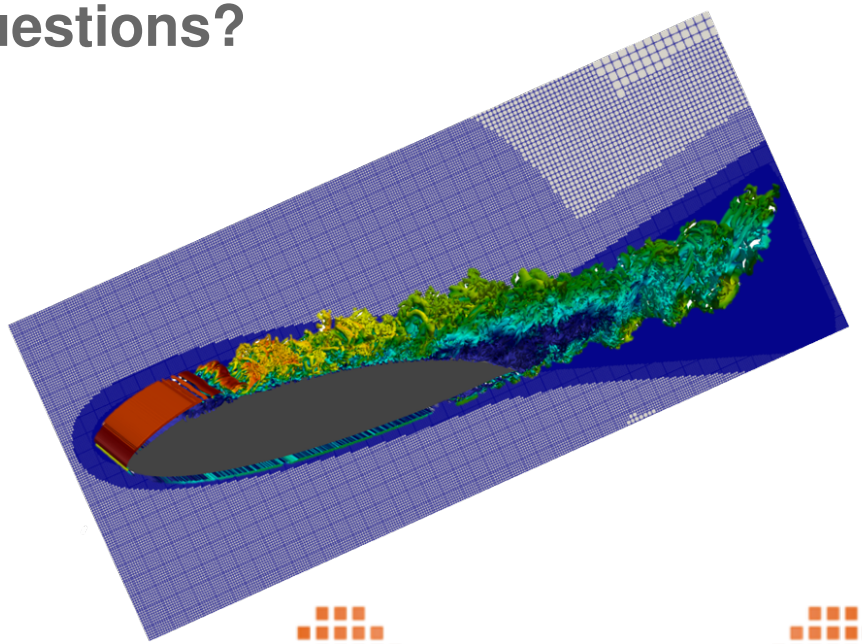


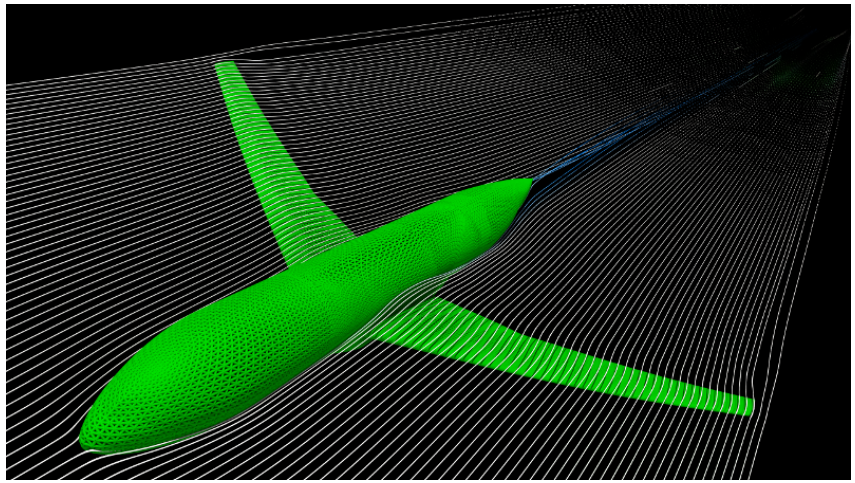


Outlook

- FlowSimulator CFD-CSM pipelines
  - Grid deformation & repair
- CFD solver TRACE
- B2000++Pro

Questions?





VAMPIR

Score-P

Scalable performance measurement infrastructure for parallel codes

Gist

- ↗ HPCs are getting larger
- ⇒ Higher scalability needed for HPC codes
- ⇒ In-depth analysis of codes required
- ↗ DLR HPC competence center can help

DLR