# Case studies of OpenMP & MPI

Matthias Möller
Department of Applied Mathematics
Delft University of Technology, The Netherlands

Joint work with Hugo Verhelst & Roel Tielen

## About

- Diploma in Mathematics from TU Dortmund, DE (2003)
- PhD in Mathematics from TU Dortmund, DE (2008)
- Associate Professor of Numerical Analysis, TU Delft

**Research interests**

- Numerical simulations and optimization of PDE problems
- Quantum computing and high-performance computing
- Scientific machine learning

## Today’s talk

- Case studies of using OpenMP / MPI
  - Brute-force QUBO sampler
  - Parallel-in-time method in G+Smo
  - Parallel Arc-Length method in G+Smo

# Brute-force QUBO sampler

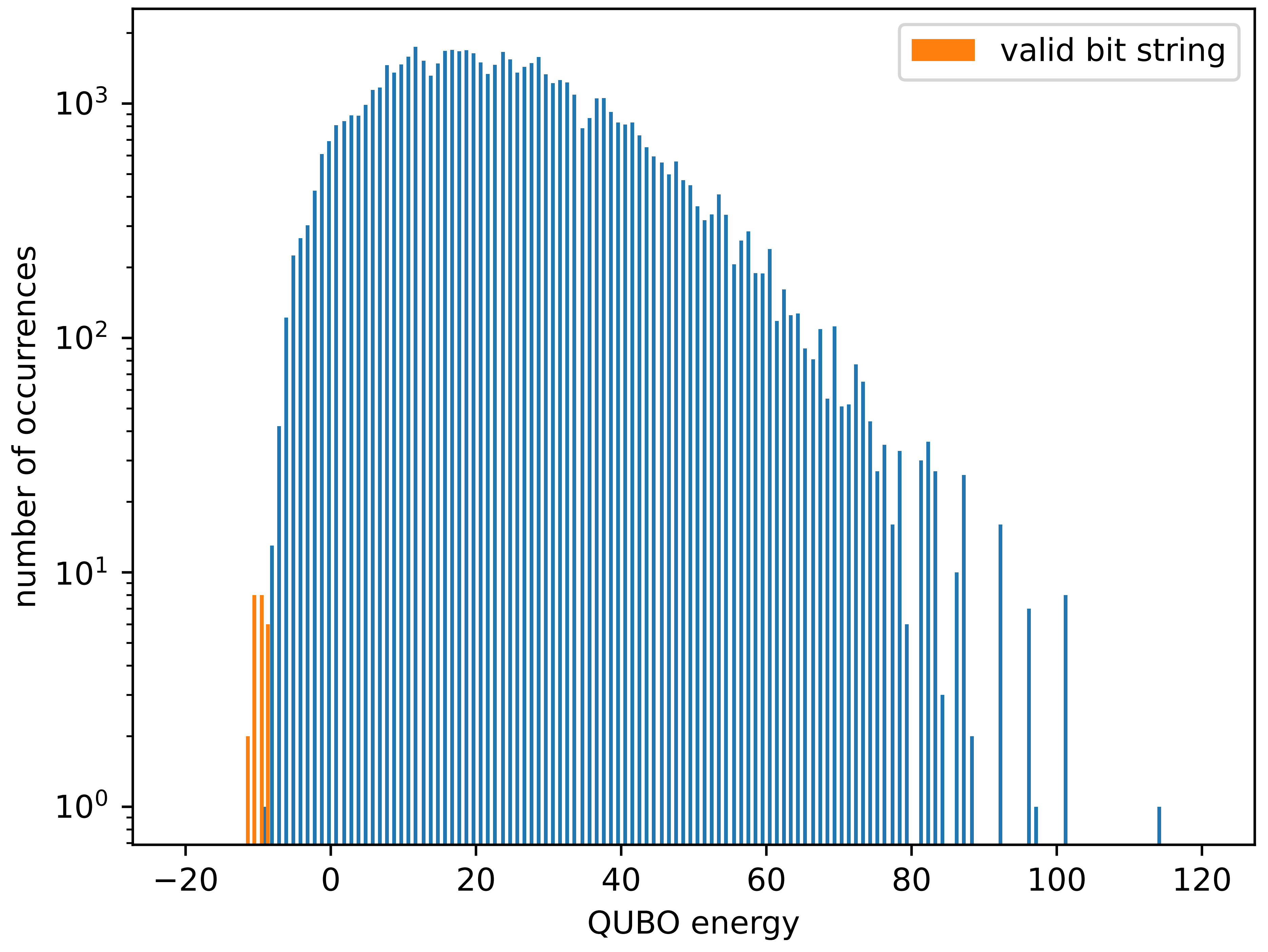
- **Problem**: Given a symmetric real-valued matrix $\mathbf{Q} \in \mathbb{R}^{n\times n}$ find a bit string $\mathbf{x} \in \{0,1\}^n$ such that

$$\mathbf{x}^* = \text{minarg}_{\mathbf{x}\in\{0,1\}^n} \mathbf{x}^\top \mathbf{Q} \mathbf{x} \quad \text{or}$$
$$e^* = \text{min}_{\mathbf{x}\in\{0,1\}^n} \mathbf{x}^\top \mathbf{Q} \mathbf{x}$$

- **Challenge**: There are $2^n$ different bit strings that need to be tested to find the *global* minimum

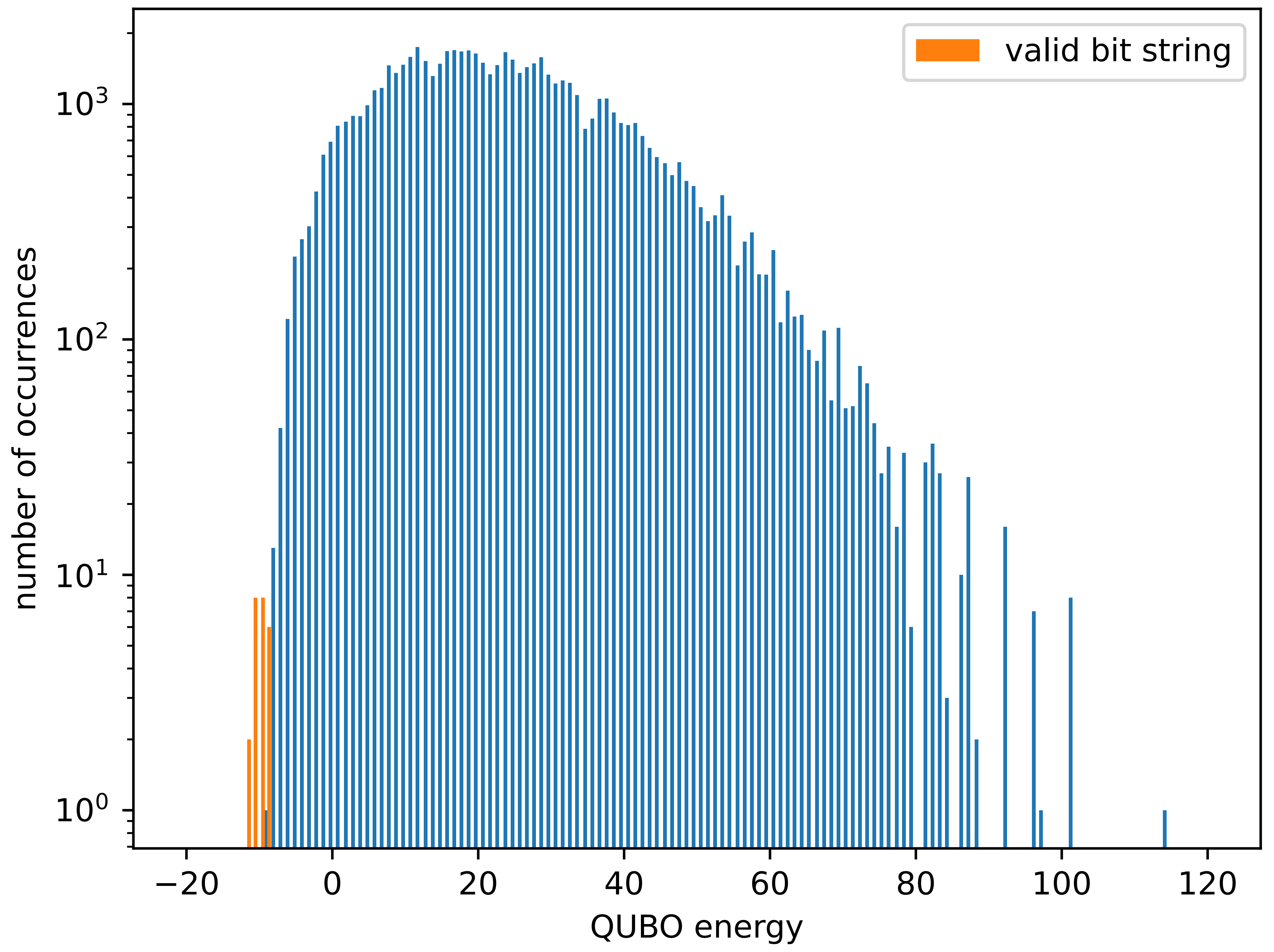
- **Approach**: Quantum annealers (D-Wave) are designed to solve this problem efficiently.

  However, for developing QUBO formulations we need an efficient brute-force sampler that can produce the full energy landscape efficiently.



# Brute-force QUBO sampler

- **Data structures**:
  - $H$: histogram($E_{\text{min}}, E_{\text{max}}, N_{\text{bins}}$)
  - $S$: array of $N_{\text{best}}$ samples
- **Sequential algorithm**:
  - For $i = 0,\ldots,2^n - 1$
    - create $\mathbf{x}_i \in \{0,1\}^n$ and compute $e_i = \mathbf{x}_i^\top \mathbf{Q} \mathbf{x}_i$
    - increment the counter of the corresponding "energy bin" by one and insert $e_i$ into array of best samples if appropriate (sorting!)



## Implementation details

- C++20 compute kernel with PyBind11 wrapper
- Linear algebra library
  - $\mathbf{Q}$: `Eigen::SparseMatrix<T, Eigen::ColMajor>` or `Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic>`
  - $\mathbf{x}$: `Eigen::Vector<T, Eigen::Dynamic>`
- MPI parallelization
  - Each rank has its own histogram $H$ and array $S$, computes

    ```
    T energy = (x.cwiseProduct(Q*x)).colwise().sum()[0];
    ```

    and updates $H$ and potentially $S$

## Implementation details

```
struct Histogram {
  Histogram(…) {…}
  std::vector<std::size_t> bins;
  std::vector<double>      values;
};

// reduce global histogram from all MPI processes
Histogram global_hist(binCount, minValue, maxValue);
MPI_Allreduce(hist.bins.data(), global_hist.bins.data(),
              binCount, MPI_UNSIGNED_LONG_LONG, MPI_SUM, MPI_COMM_WORLD);

// we don’t have to allreduce hist.values because they are the same for all
// copies and are set by the constructor
```

## Implementation details

```
struct Samples {
  Samples(…) {…}
  std::vector<std::pair<double, std::size_t>> samples;
};

MPI_Aint baseaddr, addr, displacement[2];
MPI_Get_address (&samples.samples.data()->first, &baseaddr);
MPI_Get_address (&samples.samples.data()->second, &addr);
displacement[0] = 0; displacement[1] = addr - baseaddr;

MPI_Datatype datatype[2];
datatype[0] = MPI_DOUBLE; datatype[1] = MPI_UNSIGNED_LONG_LONG;

int blocklength[2]; blocklength[0] = 1; blocklength[1] = 1;

MPI_Datatype MPI_PAIR;
MPI_Type_create_struct(2, blocklength, displacement, datatype, &MPI_PAIR);
MPI_Type_commit(&MPI_PAIR);
```

## Implementation details

```
// reduce samples from all MPI processes
Samples global_samples(nprocs * nsamples);
MPI_Allgather(samples.samples.data(), nsamples, MPI_PAIR,
              global_samples.samples.data(), nsamples, MPI_PAIR, MPI_COMM_WORLD);

// sort and resize samples from all MPI processes
std::sort(global_samples.samples.begin(), global_samples.samples.end(),
          [](const auto& lhs, const auto& rhs)
            { return lhs.first < rhs.first; });

while (global_samples.samples.size() > nsamples)
  global_samples.samples.pop_back();
```

## Scalability: N=26, 67.108.864 bit strings, 15% fill

| MPI | Runtime | Parallel runtime |
|---|---|---|
| 1 | 19.57 | 19.57 |
| 2 | 7.61 - 9.52 | 18.95 |
| 4 | 3.80 - 4.75 | 18.70 |
| 8 | 1.89 - 2.37 | 18.24 |
| 16 | 0.94 - 1.20 | 17.58 |
| 32 | 0.47 - 0.60 | 17.31 |
| 64 | 0.23 - 0.24 | 19.55 |
| 128 | 0.11 - 0.13 | 54.72 |
| 192 | 0.11 - 0.12 | 45.55 |

Python kills performance

- Long startup times (`import`)
- GIL hinders effective OpenMP

## Lessons learned

- Close to optimal scaling up to 128 cores per node for compute-bound nearly data-less problem
- User-defined MPI datatypes can be used to perform MPI operations on C++ datatypes
- DON’T USE PYTHON FOR PARALLEL PROGRAMMING

**Not discussed here**

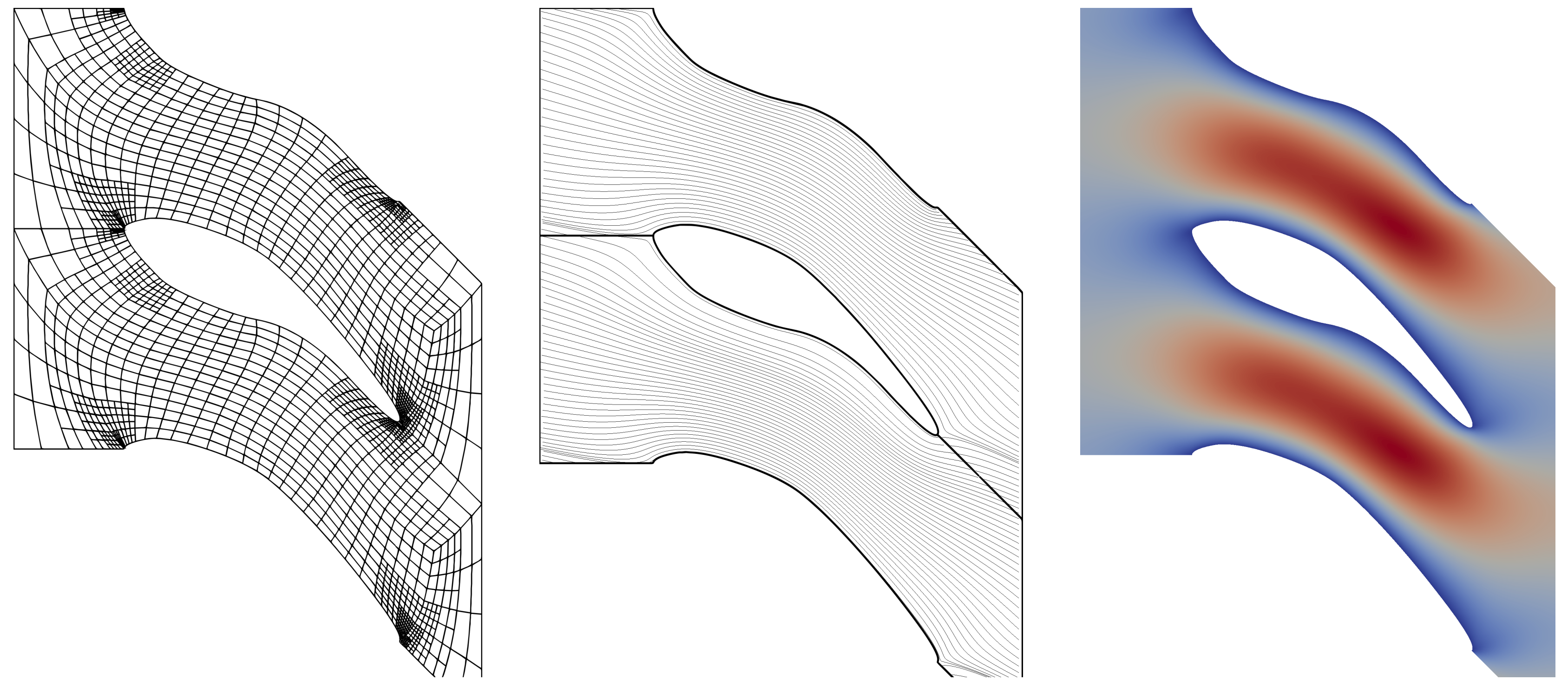
- Since the QUBO matrix is the same for all processes, we implemented a MPI+OpenMP variant with 1-2 MPI process(es) per node and multiple OpenMP threads per MPI process (user-defined OpenMP reduction, atomic updates of $H$ and $S$). Works outside Python but is limited by GIL.

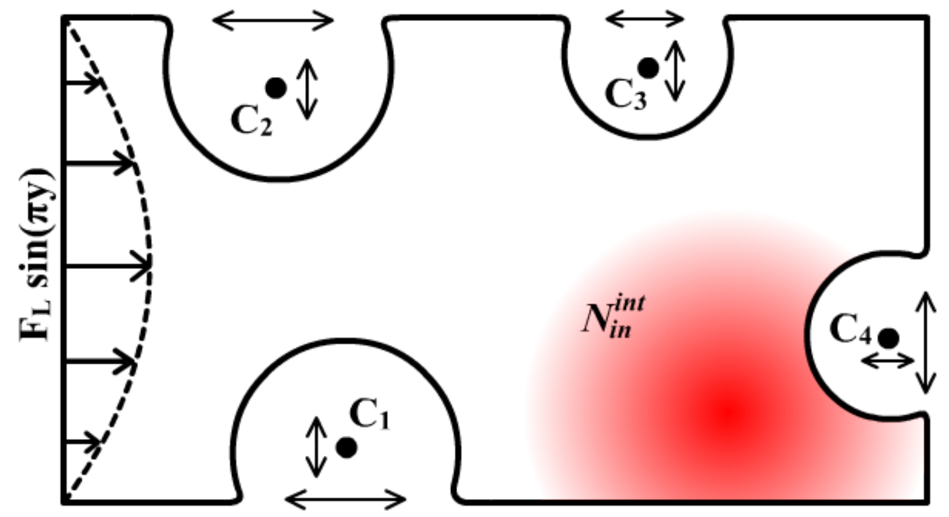
## G+Smo - Geometry plus Simulation Modules

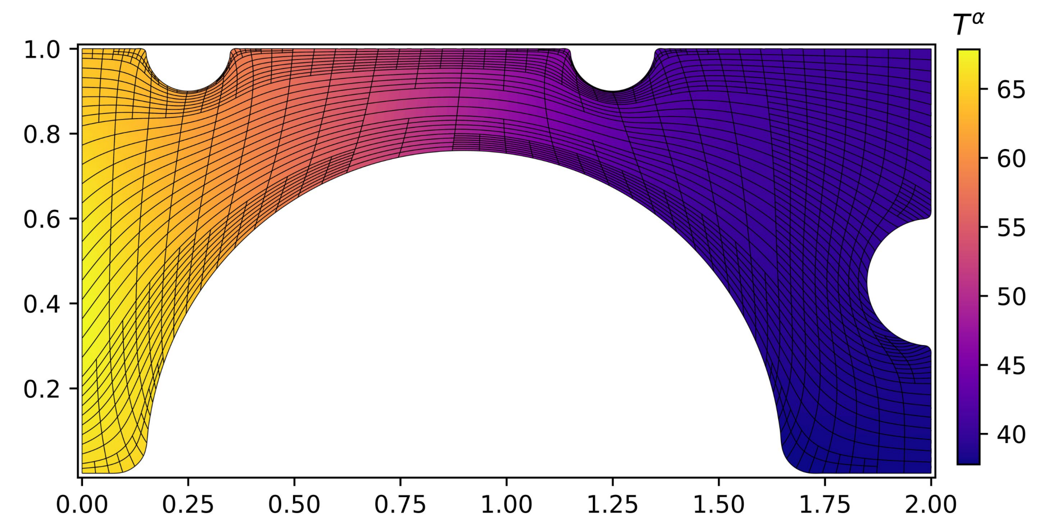
- Open-source (MLP-2.0) isogeometric analysis library written in C++11 on top of the Eigen library
- Developers & users: Inria, TU Delft, JKU, RICAM, UoFlorence, MTU AeroEngines, Vtech CMCC, …
- Features:
  - OpenMP parallelization (WIP), MPI parallelization (demonstrator apps + external libraries)
  - Wrappers for Python (PyBind11), Julia (WIP), Matlab (WIP)
  - External libraries: OpenNurbs, Pardiso, Trilions, Spectra, CoDiPack, …
  - Import/export formats: XML, VTK, 3dm, …

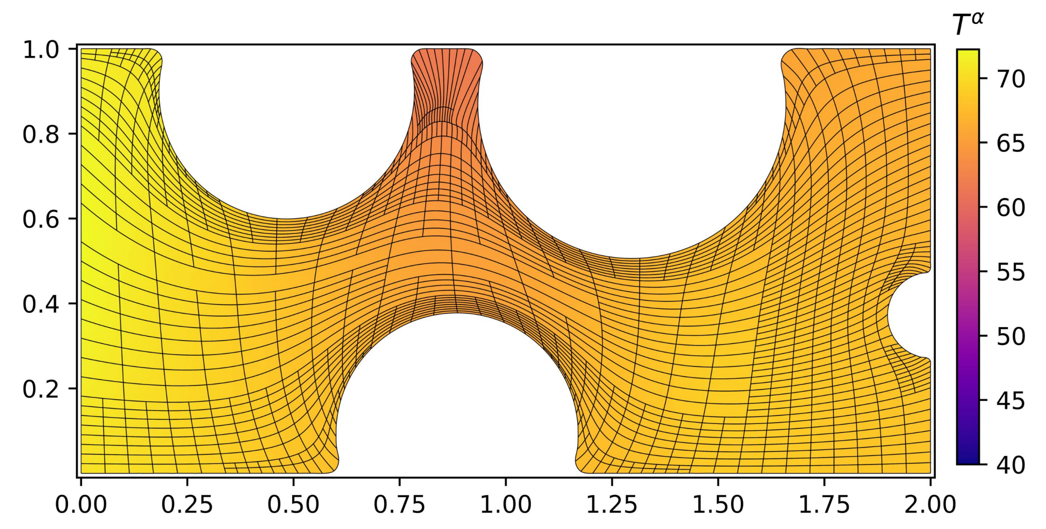
## Typical applications

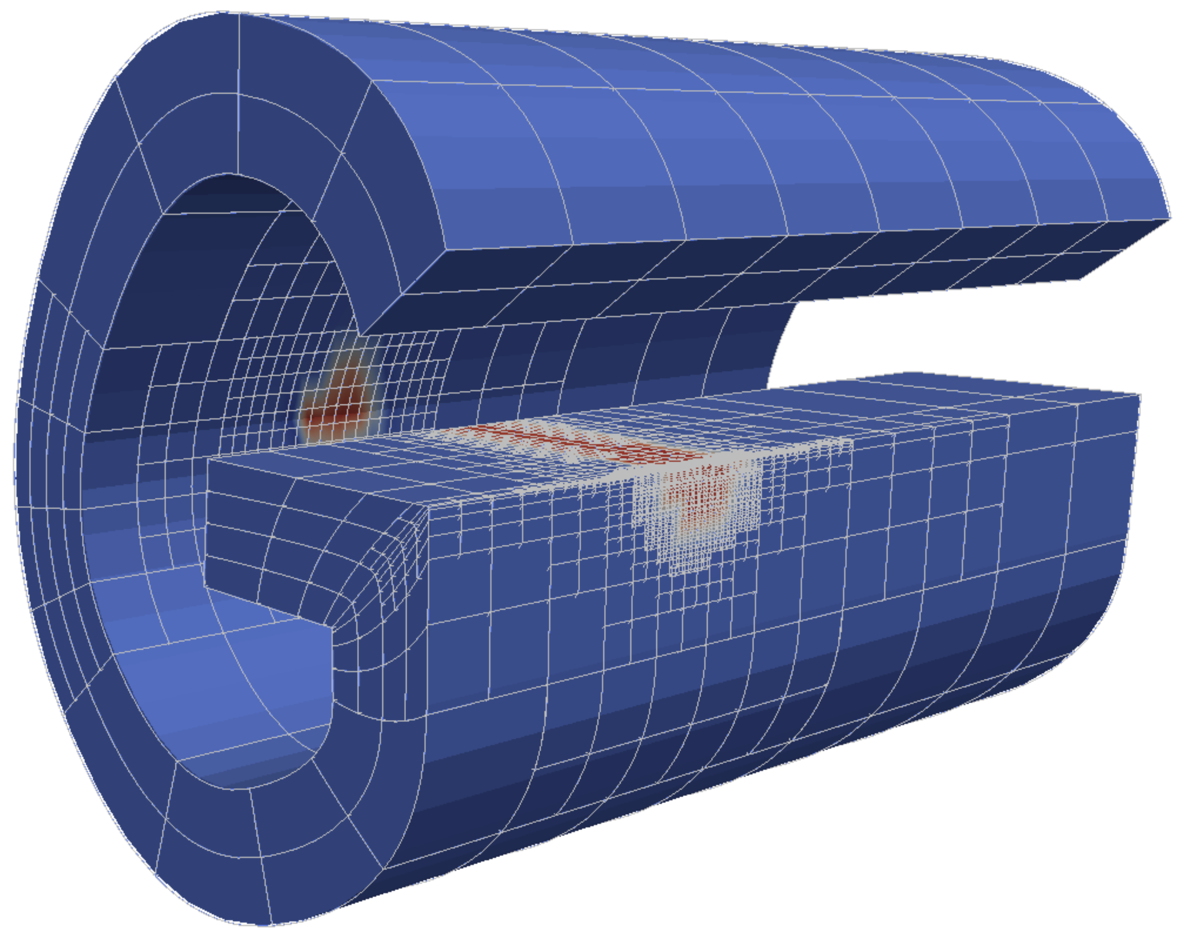
- Geometric modelling with adaptive splines
- Simulation of linear/nonlinear PDE problems
- PDE-constrained shape/topology optimization











## Sequential time integrators

- Model problem

$$\frac{du(t)}{dt} = A(u(t), t), \quad u(t=0) = u^0$$

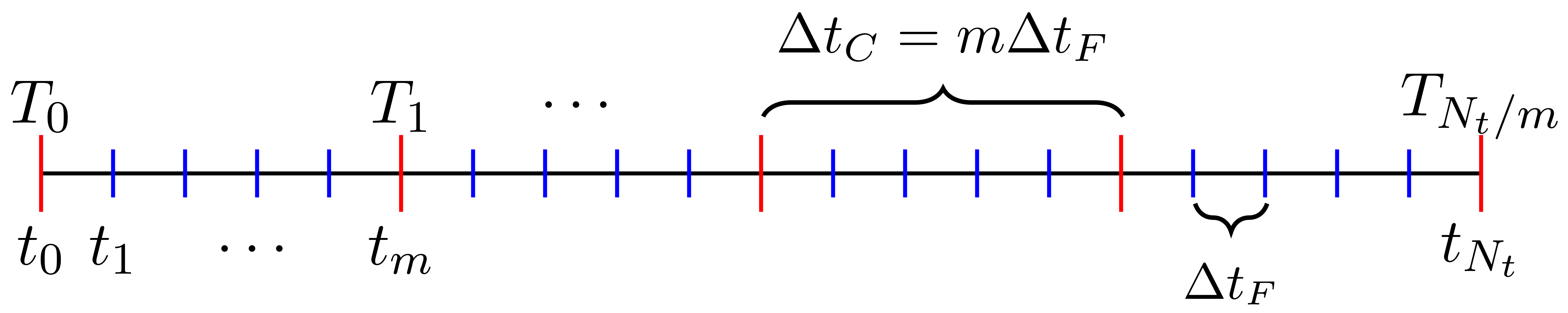
- Explicit time integrators

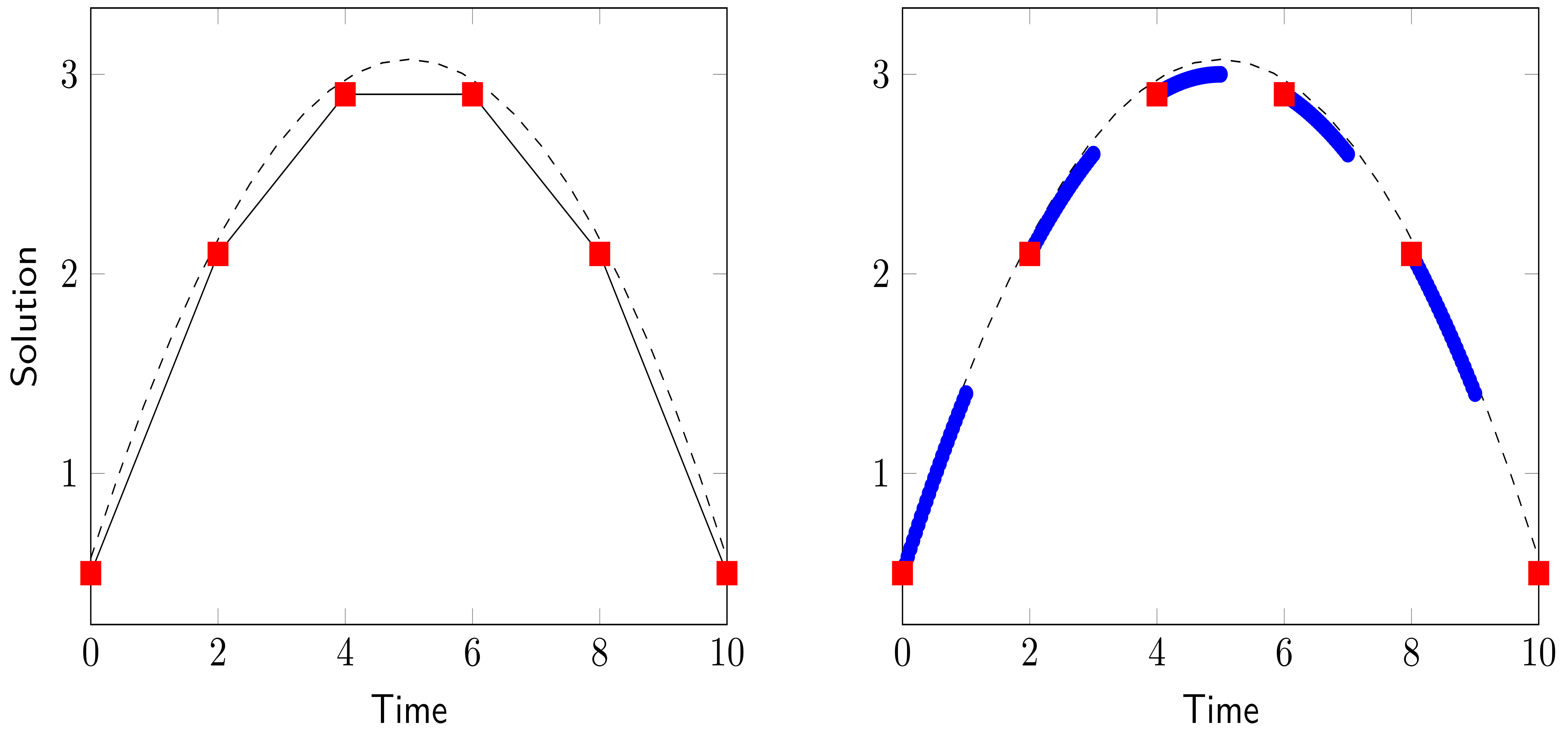
$$\frac{u^{n+1} - u^n}{\Delta t} = A(u^n, t^n) \quad \Rightarrow \quad u^{n+1} = u^n + \Delta t A(u^n, t^n), \quad u^n := u(t^n)$$

- Implicit time integrators

$$\frac{u^{n+1} - u^n}{\Delta t} = A(u^{n+1}, t^{n+1}) \quad \Rightarrow \quad u^{n+1} - \Delta t A(u^{n+1}, t^{n+1}) = u^n$$

## Parallel-in-time integrators





## Sketch of the parallel-in-time algorithm

- Writing out the two-level time integration scheme $[M + \Delta t_F K]u^{n+1} = Mu^n + f$ for all time levels yields

$$\begin{bmatrix} I & & & \\ -\Psi M & I & & \\ & \ddots & \ddots & \\ & & -\Psi M & I \end{bmatrix} \begin{bmatrix} u^0 \\ u^1 \\ \vdots \\ u^N \end{bmatrix} = \Delta t_F \begin{bmatrix} \Psi f \\ \Psi f \\ \vdots \\ \Psi f \end{bmatrix} \quad \text{with} \quad \Psi = [M + \Delta t_F K]^{-1}$$

- Reordering of the system matrix (and the vectors!) into $F$ine and $C$oarse time levels yields

$$\begin{bmatrix} A_{FF} & A_{FC} \\ A_{CF} & A_{CC} \end{bmatrix} = \begin{bmatrix} I_F & 0 \\ A_{CF}A_{FF}^{-1} & I_C \end{bmatrix} \begin{bmatrix} A_{FF} & 0 \\ 0 & S \end{bmatrix} \begin{bmatrix} I_F & A_{FF}^{-1}A_{FC} \\ 0 & I_C \end{bmatrix} \quad \text{with} \quad S = A_{CC} - A_{CF}A_{FF}^{-1}A_{FC}$$

# Sketch of the parallel-in-time algorithm

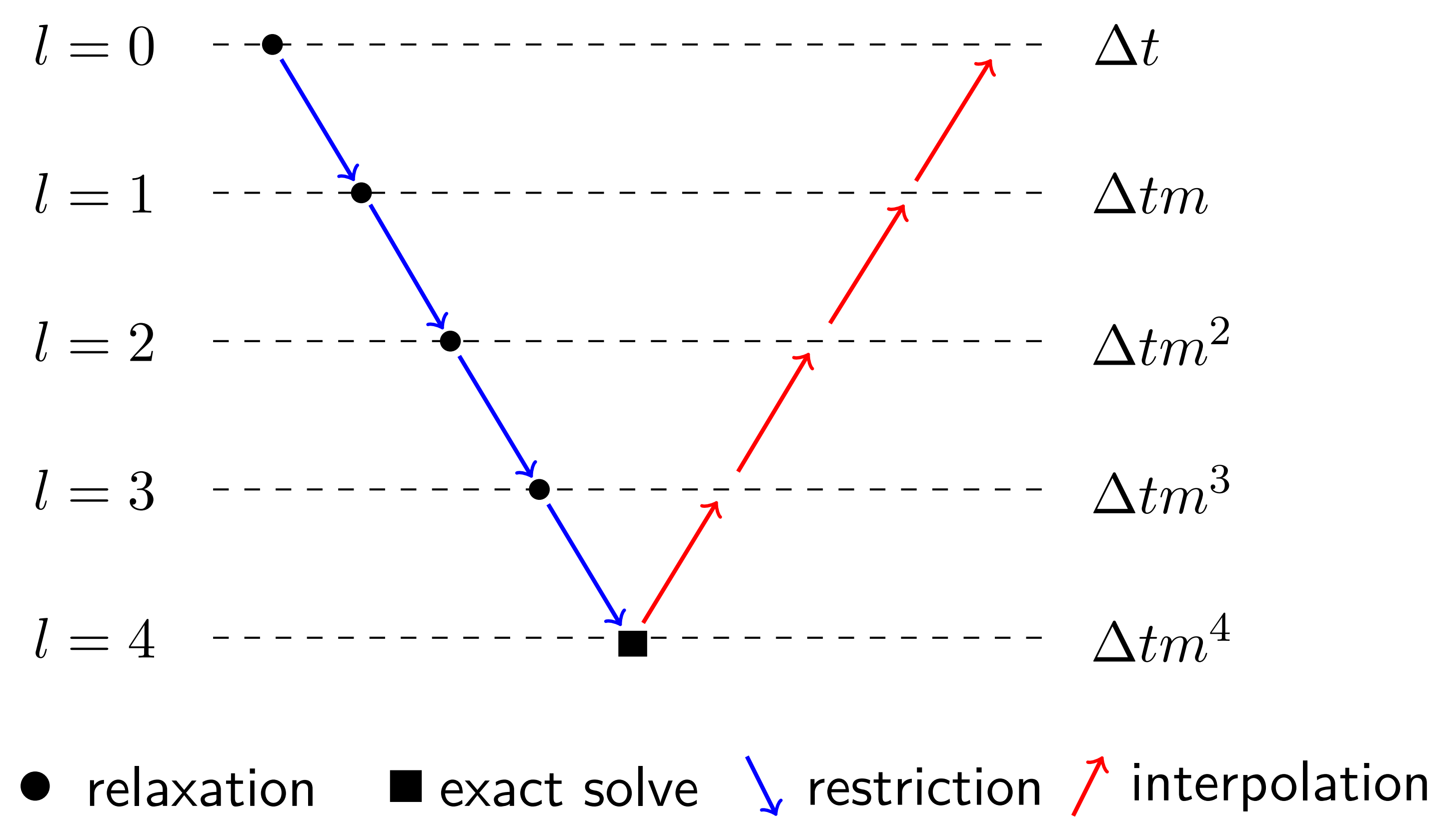
- Approximation of the **Schur complement** matrix

$$S = \begin{bmatrix} I & & & \\ -(\Psi M)^m & I & & \\ & \ddots & \ddots & \\ & & -(\Psi M)^m & I \end{bmatrix} \approx \begin{bmatrix} I & & & \\ -\Phi M & I & & \\ & \ddots & \ddots & \\ & & -\Phi M & I \end{bmatrix}$$

- Approximate **coarse integrator**

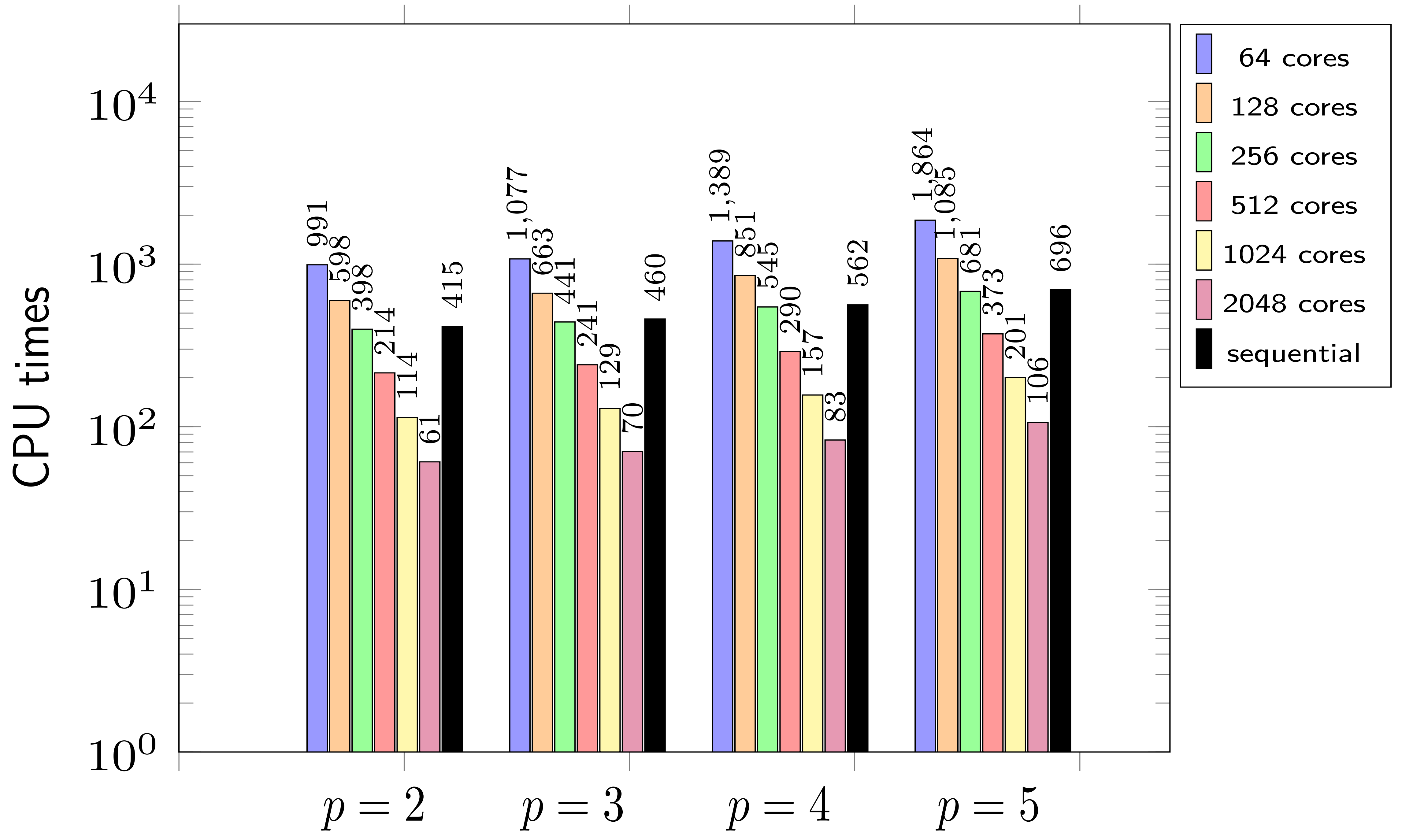
$$\Phi = \left[M + \Delta t_C\, K\right]^{-1}$$

- Repeat this two-level recursion to obtain **MGRIT (multi-grid-in-time)**



## Strong scaling

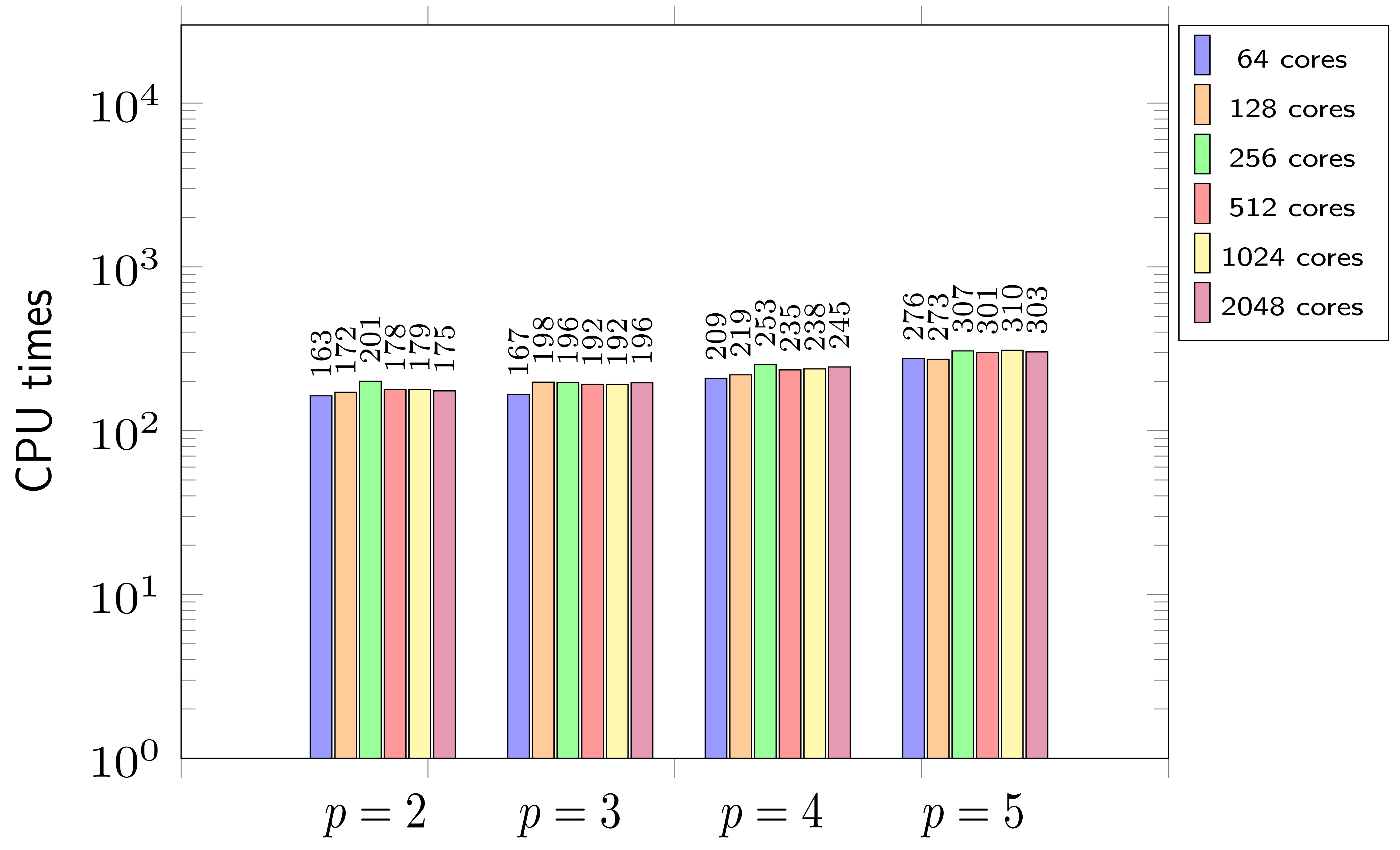
- 2d heat equation with $h = 2^{-6}$ spatial resolution solved for $N_t = 10.000$ time steps using the backward Euler scheme and IGA discretisation on 128 Xeon Gold 6130 CPUs (2.10GHz, 96GB, 16 cores)



R. Tielen, M. Möller, and C. Vuik (2022) Combining p-multigrid and multigrid reduction in time methods to obtain a scalable solver for isogeometric analysis, SN Appl. Sci. 4

# Weak scaling

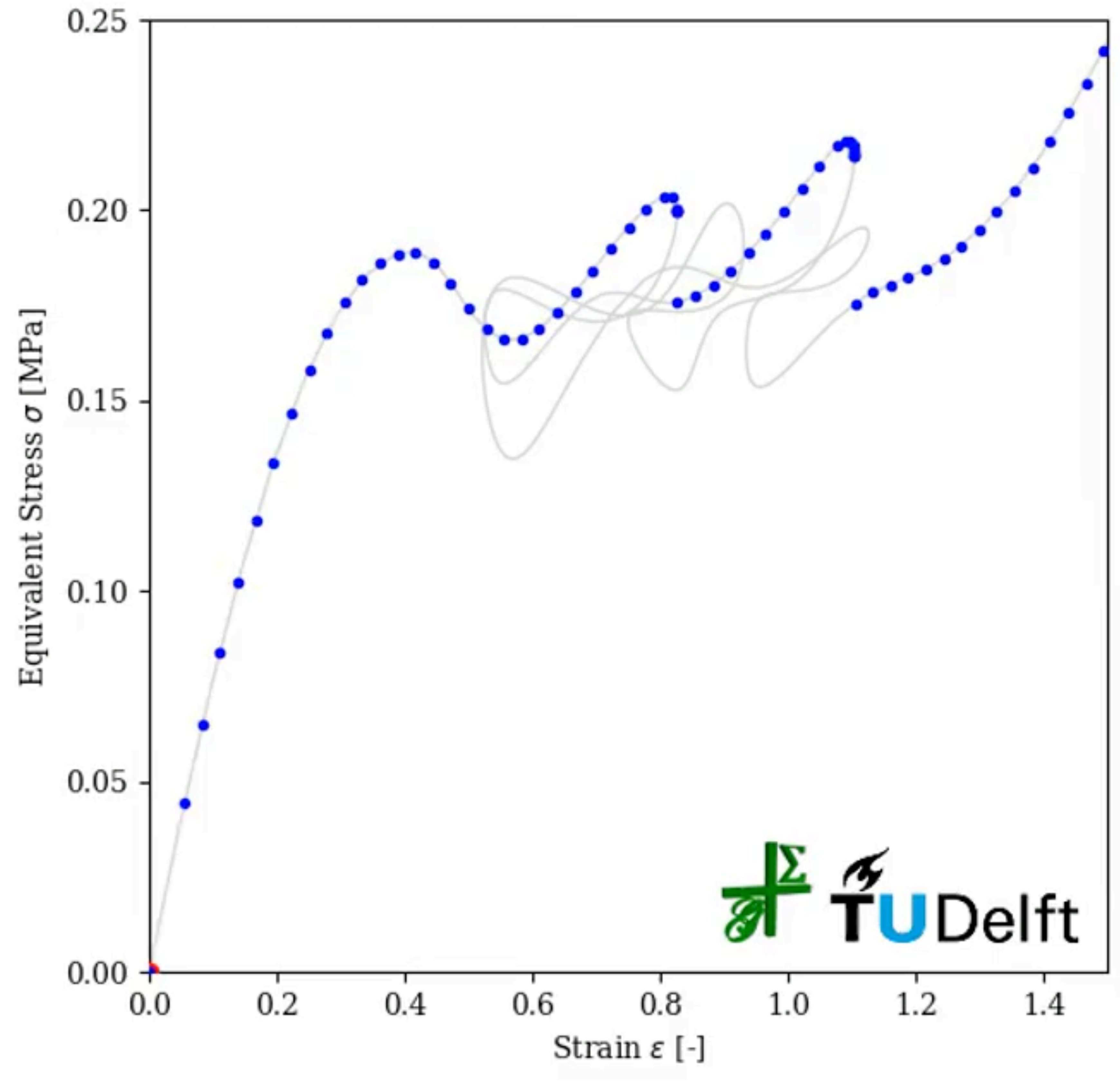
- 2d heat equation with $h = 2^{-6}$ spatial resolution solved for $N_t = \#\text{cores}/64 \cdot 1.000$ time steps using the backward Euler scheme and IGA discretisation on 128 Xeon Gold 6130 CPUs (2.10GHz, 96GB, 16 cores)

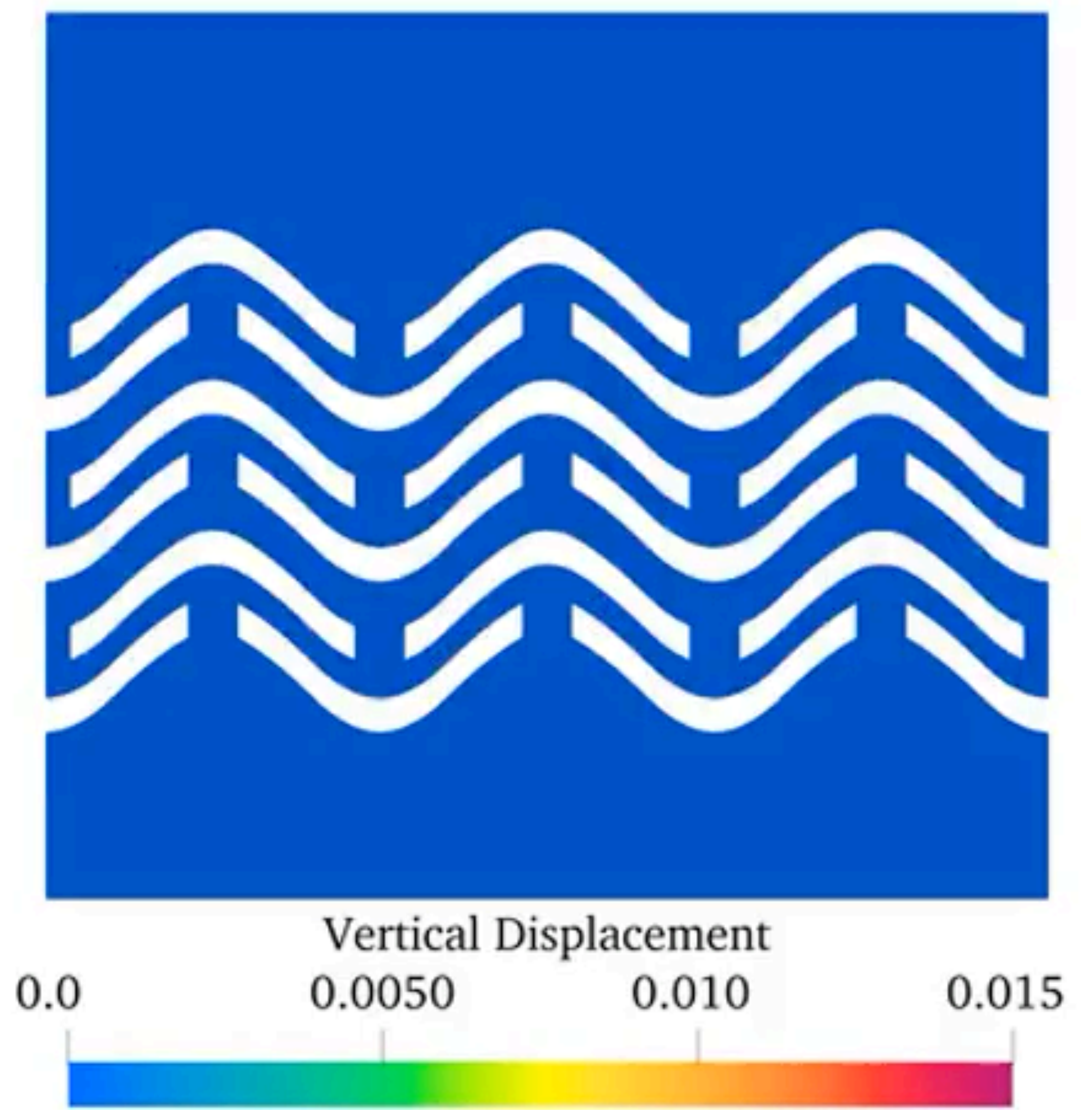


## Lessons learned

- Sequential processes like time integration can be parallelised using parallel-in-time methods
- Sufficient number of MPI processes is required to compensate the computational/mathematical overhead
- Rest of the math (and implementation) needs to be right as well
  - Large $\Delta t_C$ lead to unstable explicit integrators $\Rightarrow$ use (semi-)implicit time integrators
  - Stationary problems need to be solved efficiently (PhD Thesis by Roel Tielen)
  - It’s very difficult to utilise many-core CPUs efficiently for memory-bound problems

## Snapping meta-material simulation





H.M. Verhelst, J.H. Den Besten, and M. Möller (2024) An adaptive parallel arc-length method, Computers & Structures 296:107300

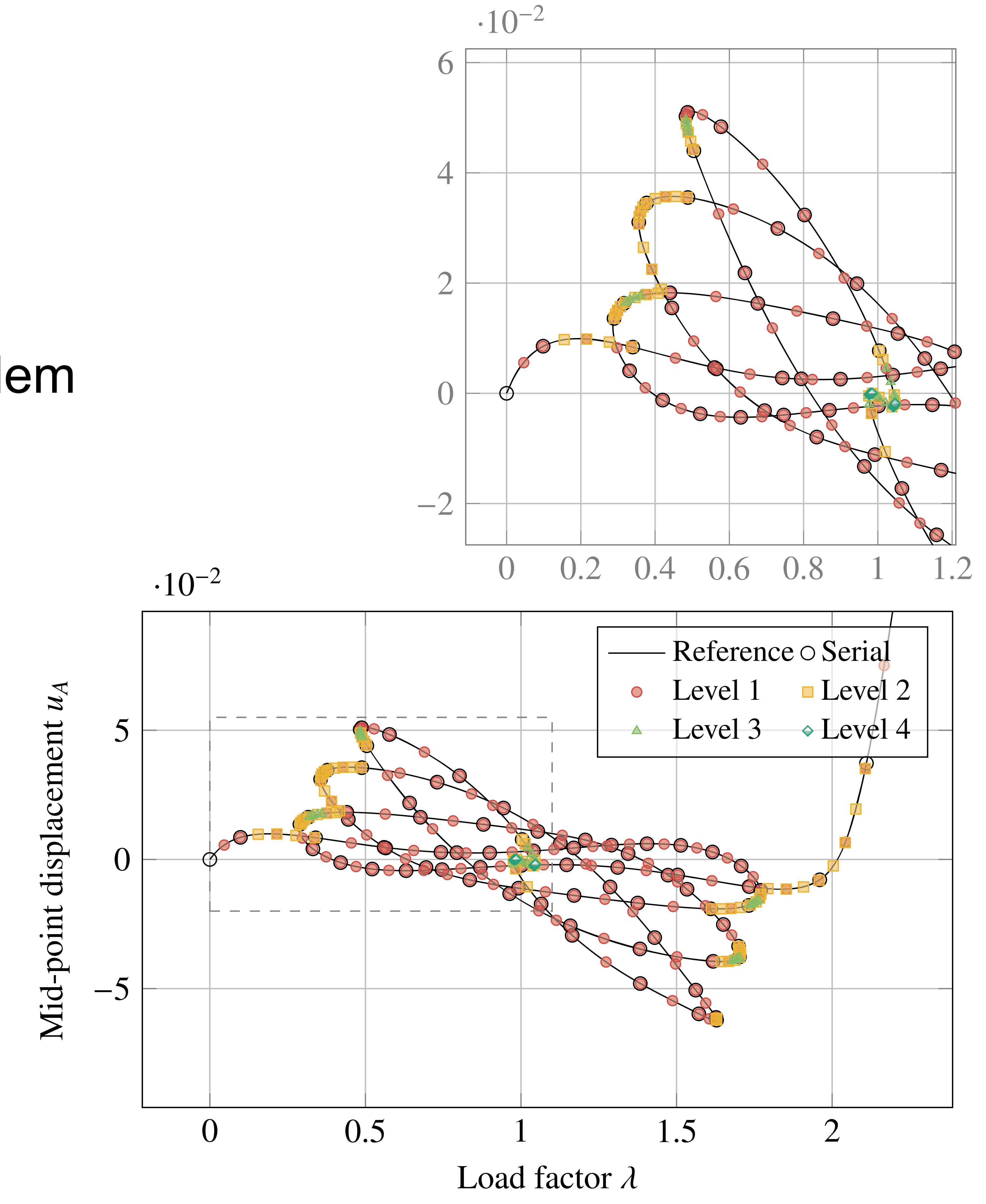
## Arc-length method (ALM)

- Nonlinear system of equations

$$\mathbf{G}(\mathbf{u},\lambda) = \mathbf{N}(\mathbf{u}) - \lambda\mathbf{P} = \mathbf{0}$$

  - $\mathbf{u}$ is the displacement vector computed by some PDE problem
  - $\mathbf{N}(\mathbf{u})$ is the vector of internal forces depending on $\mathbf{u}$
  - $\lambda$ is a scaling factor for the applied load $\mathbf{P}$

- **Task**: find the load-response curve $\{(\mathbf{u},\lambda)\ :\ \mathbf{G}(\mathbf{u},\lambda) = \mathbf{0}\}$
- **Challenges**: bifurcation points, convergence problems, find the full load-response path not a set of discrete points, …

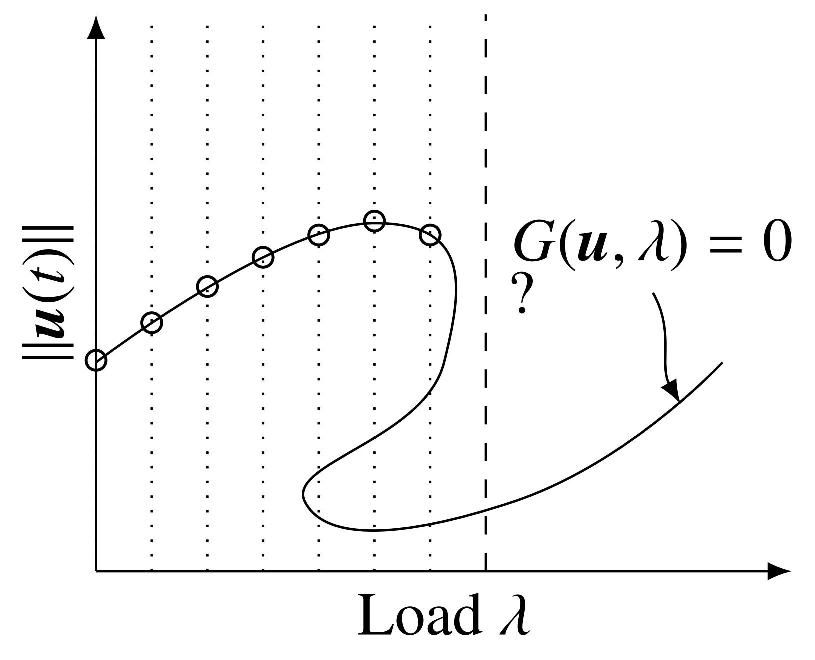


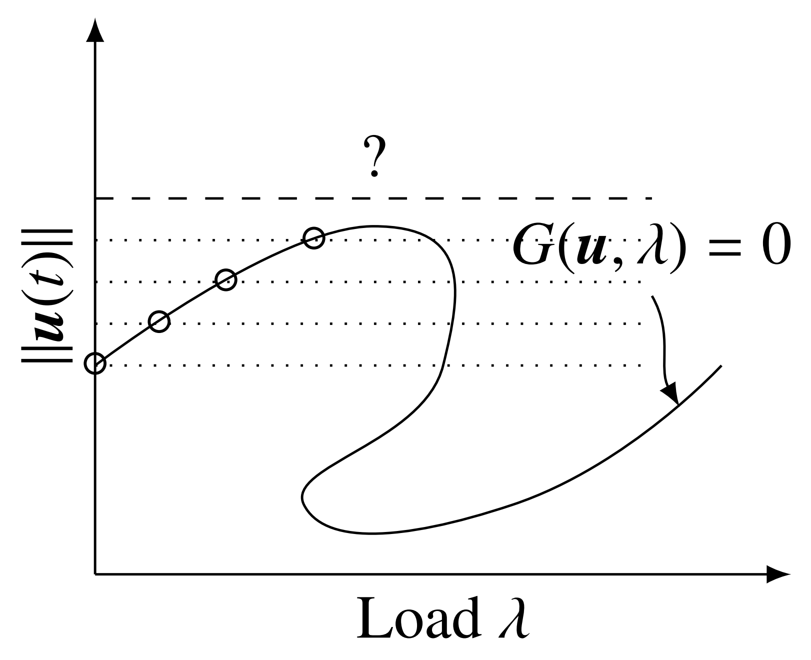
# Sequential ALM

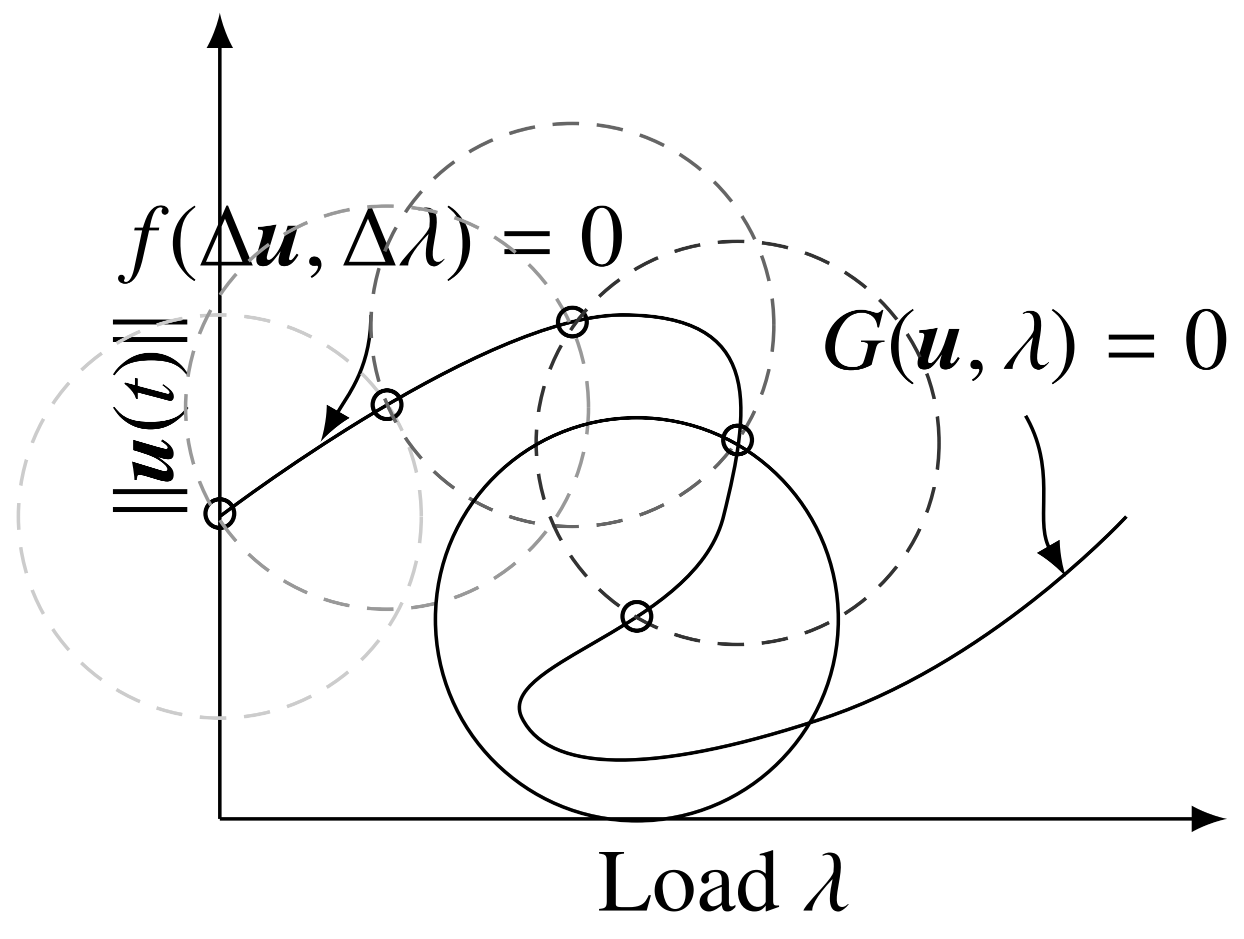
- Start with $\mathbf{w}_0 := (\mathbf{u}_0, \lambda_0)$ such that $\mathbf{G}(\mathbf{w}_0) = \mathbf{0}$ and compute the next increment $\Delta\mathbf{w}_i = (\Delta\mathbf{u}_i, \Delta\lambda_i)$ such that $\mathbf{G}(\mathbf{w}_{i+1} = \mathbf{w}_i + \Delta\mathbf{w}_i) = \mathbf{0}$ by solving the nonlinear problem using Newton's method (for $i = 1,2,\ldots$)

- **Scenario's**
  - Load control: fix $\lambda$ and compute $\mathbf{u}$
  - Displacement control: fix $\mathbf{u}$ and compute $\lambda$
  - Arc-length control: fix $\Delta\ell$ and compute $\lambda$ <u>and</u> $\mathbf{u}$ simultaneously such that

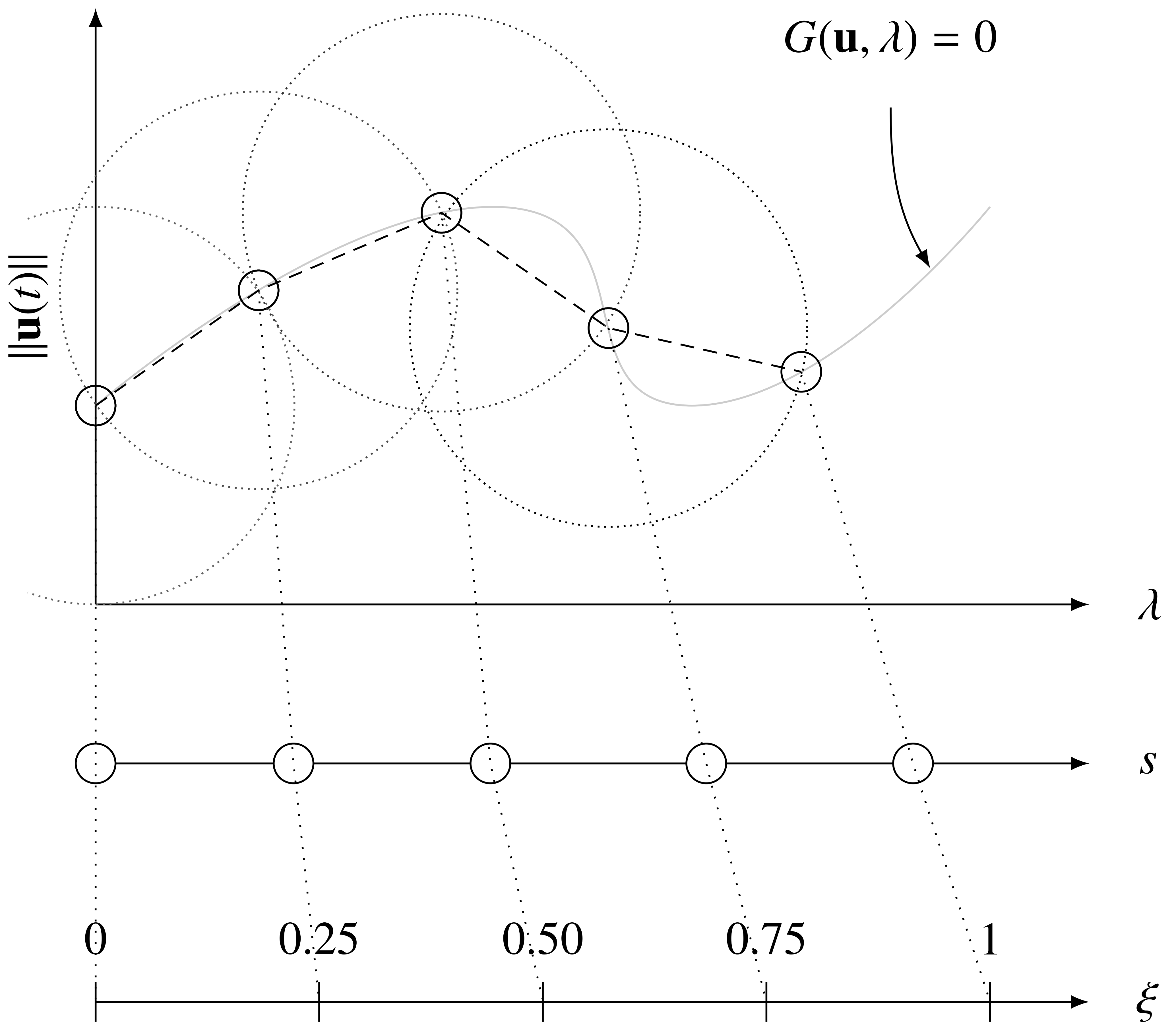
$$f(\Delta\mathbf{u}, \Delta\lambda) = \Delta\mathbf{u}^\top\Delta\mathbf{u} + \Psi^2\Delta\lambda^2\mathbf{P}^\top\mathbf{P} - \Delta\ell = 0$$



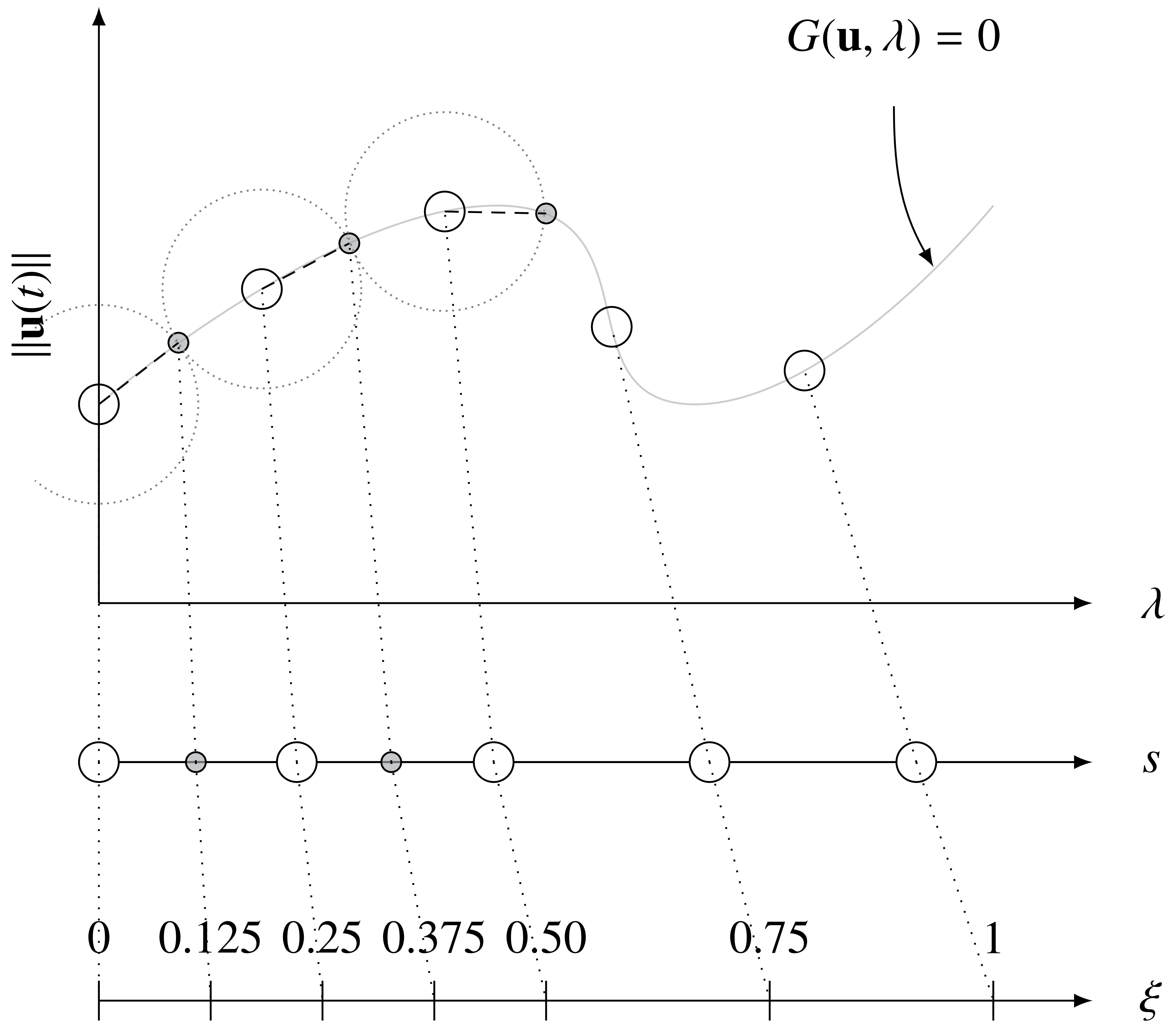



$f(\Delta\boldsymbol{u}, \Delta\lambda) = 0$
$G(\boldsymbol{u}, \lambda) = 0$
$\|\boldsymbol{u}(t)\|$
Load $\lambda$

# Adaptive parallel ALM

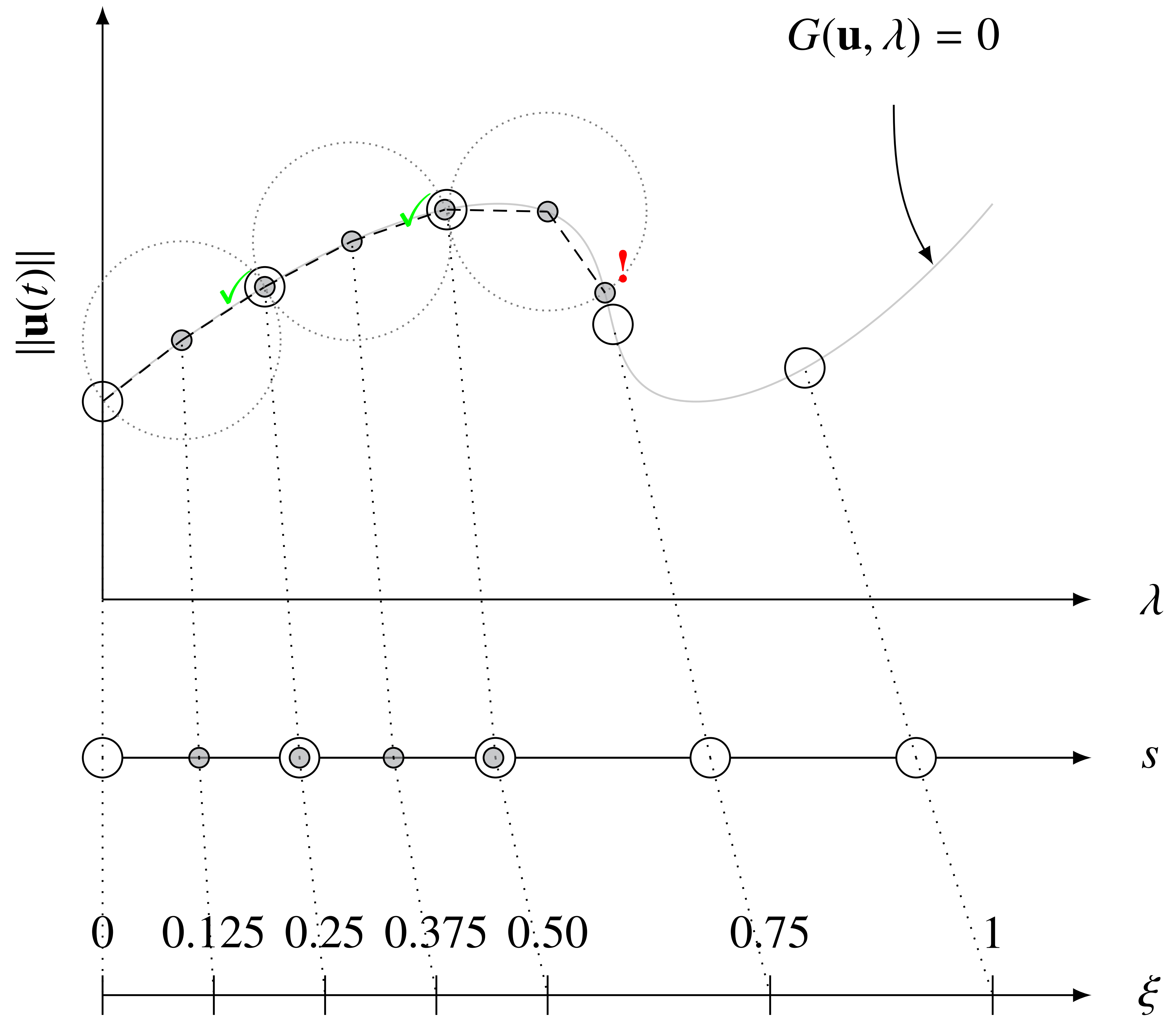


Initialisation

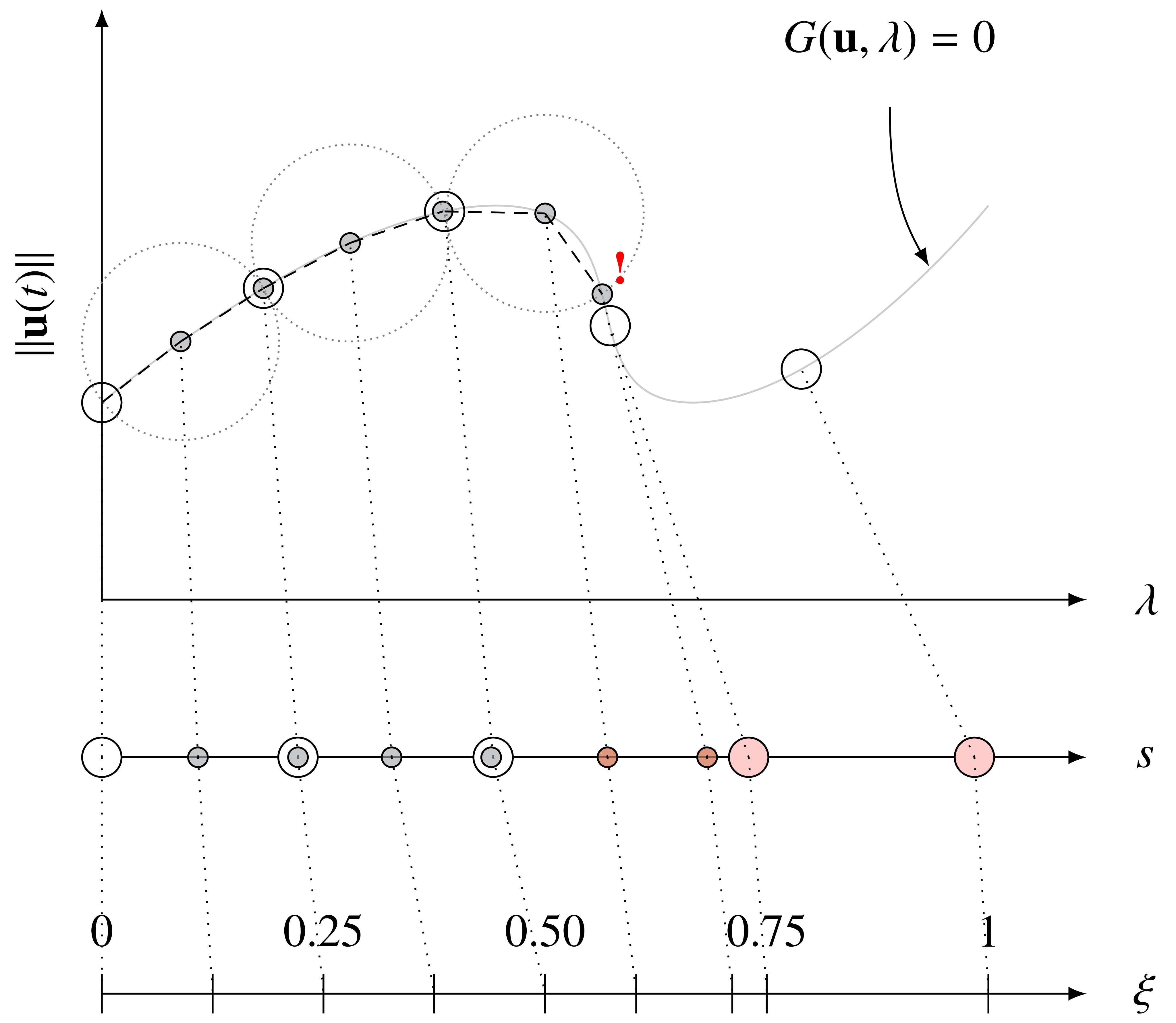


Parallel computation of subintervals

# Adaptive parallel ALM



Parallel verification of subintervals



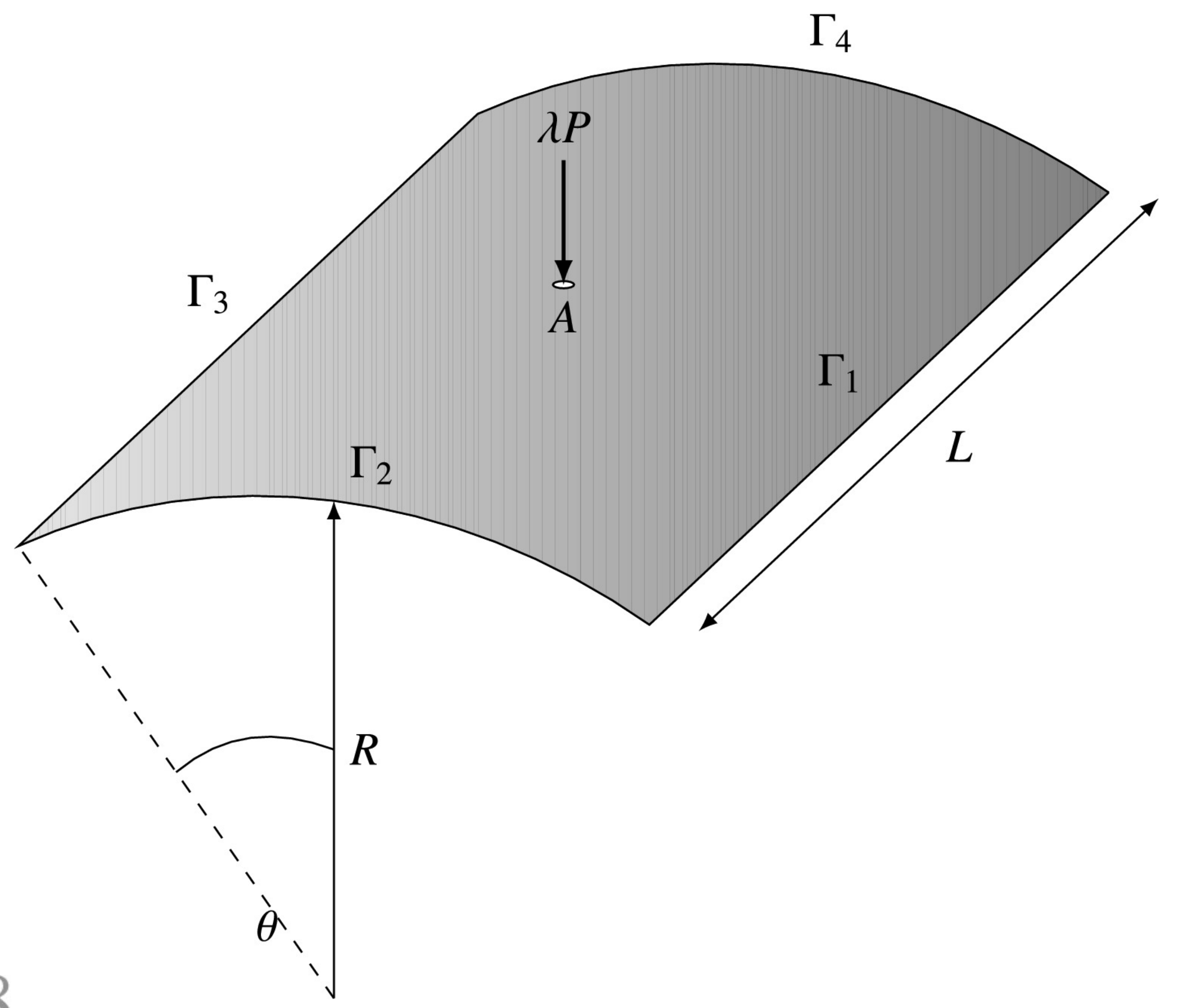
Curve-length reparameterization

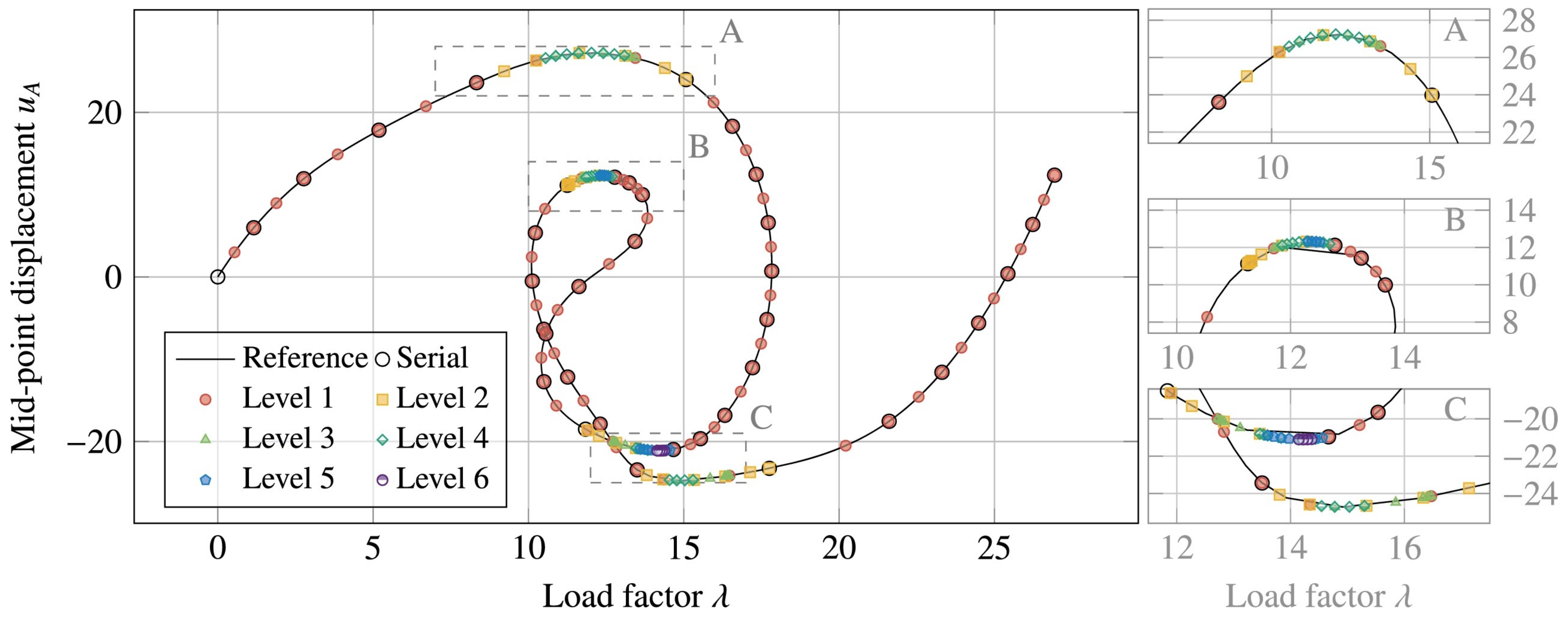
## Implementation details

- Manager (MPI rank 0)
  - Performs sequential initialisation (relaxed in the fully parallel APALM method)
  - Checks convergence criteria and sends ‘kill command’
- Pool of workers (MPI ranks 1…N)
  - Query global queue with complete ‘job description’ (problem configuration + initial guess)
  - Remove first job from queue, perform computation, add result to output list, perform validation, and add new (refined) jobs to the global job queue if needed
  - Terminate on “kill command”

## Example: collapse of a shallow roof

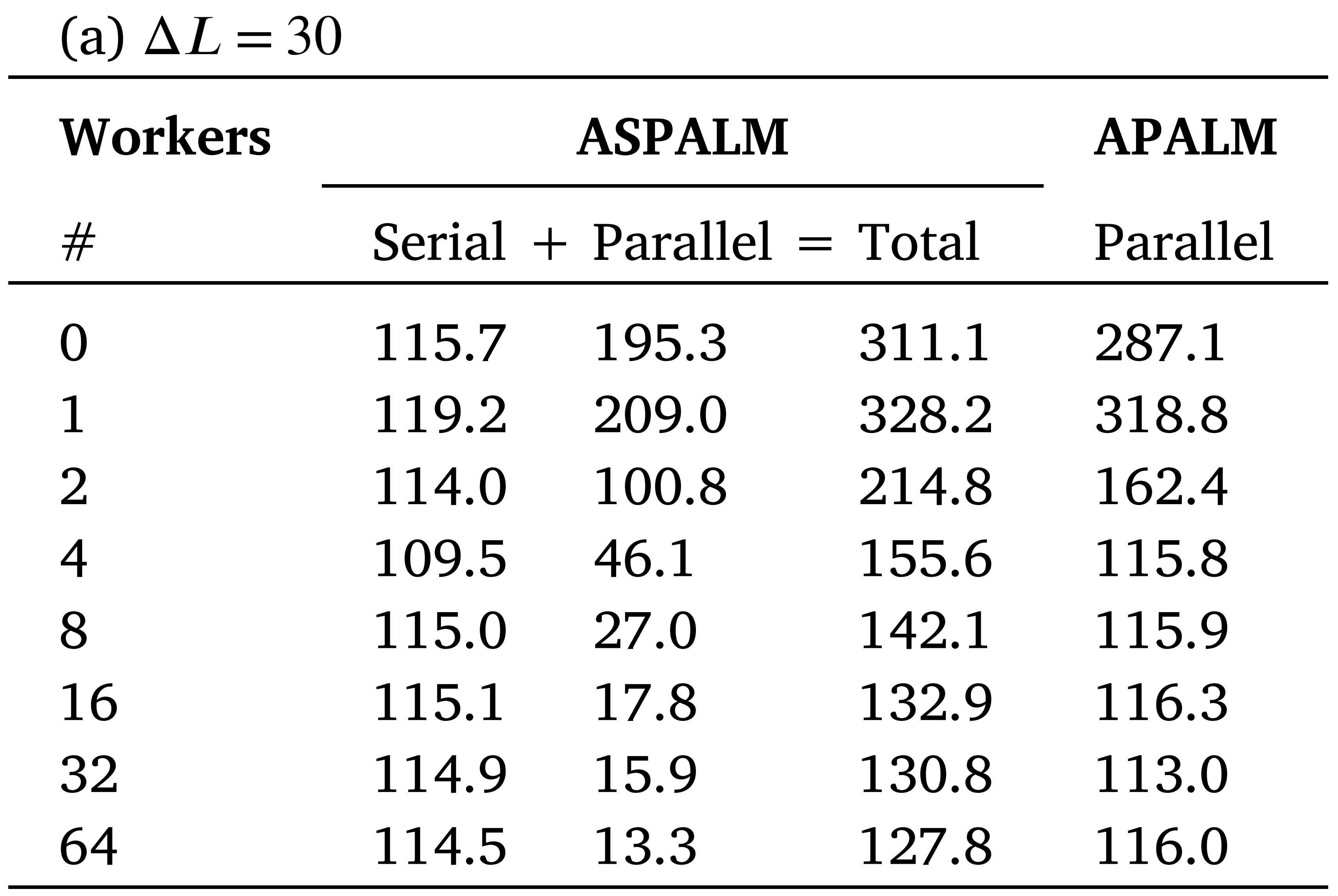
- Isogeometric Kirchhoff-Love shell model (gsKLShell extension)
- 4 x 4 NURBS elements of degree 3





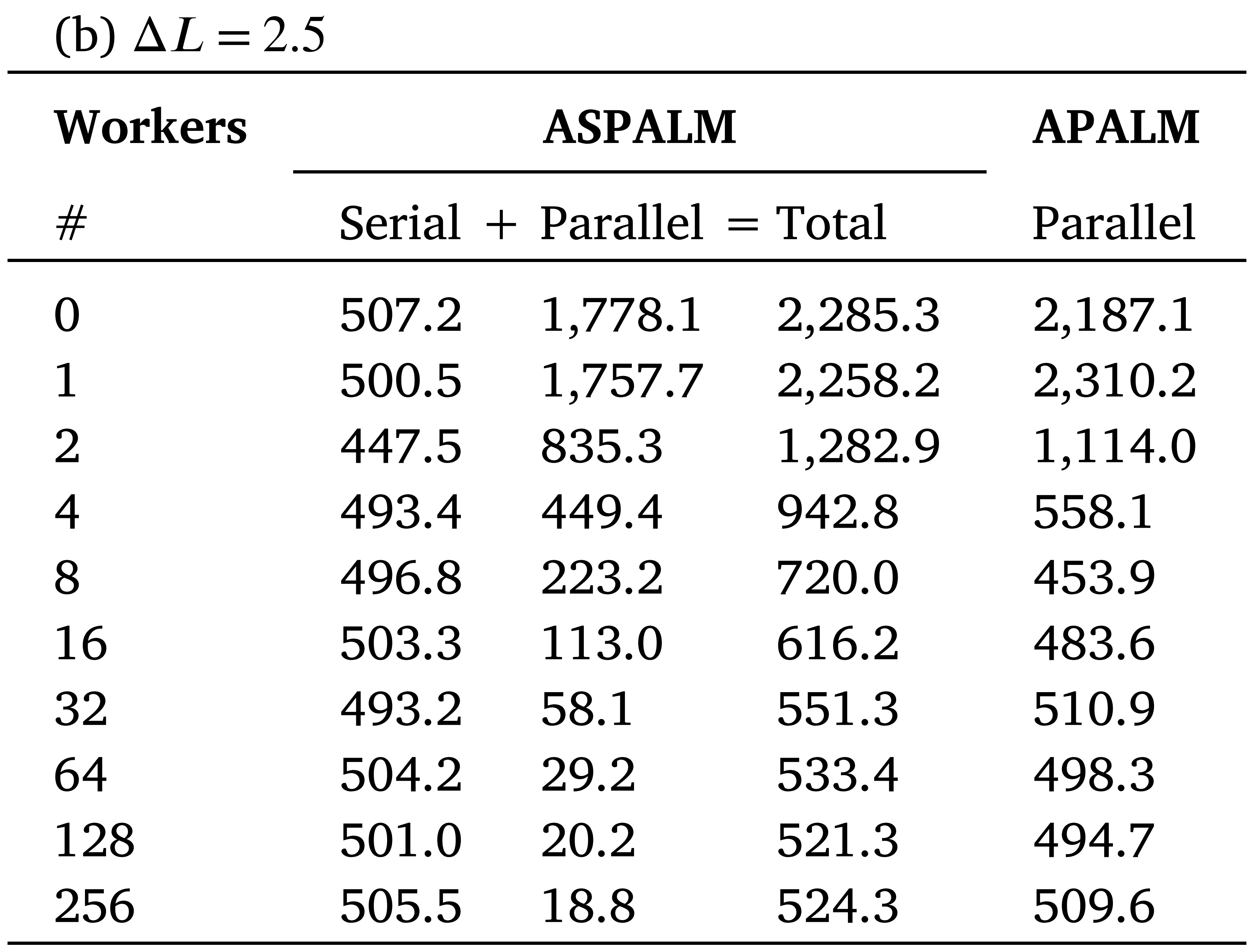
## Example: collapse of a shallow roof

- DelftBlue: Intel Xeon Gold  6248R, 24 cores @ 3.0 GHz



(a) $\Delta L = 30$

| Workers | ASPALM | | | APALM |
|---|---|---|---|---|
| # | Serial + | Parallel = | Total | Parallel |
| 0 | 115.7 | 195.3 | 311.1 | 287.1 |
| 1 | 119.2 | 209.0 | 328.2 | 318.8 |
| 2 | 114.0 | 100.8 | 214.8 | 162.4 |
| 4 | 109.5 | 46.1 | 155.6 | 115.8 |
| 8 | 115.0 | 27.0 | 142.1 | 115.9 |
| 16 | 115.1 | 17.8 | 132.9 | 116.3 |
| 32 | 114.9 | 15.9 | 130.8 | 113.0 |
| 64 | 114.5 | 13.3 | 127.8 | 116.0 |



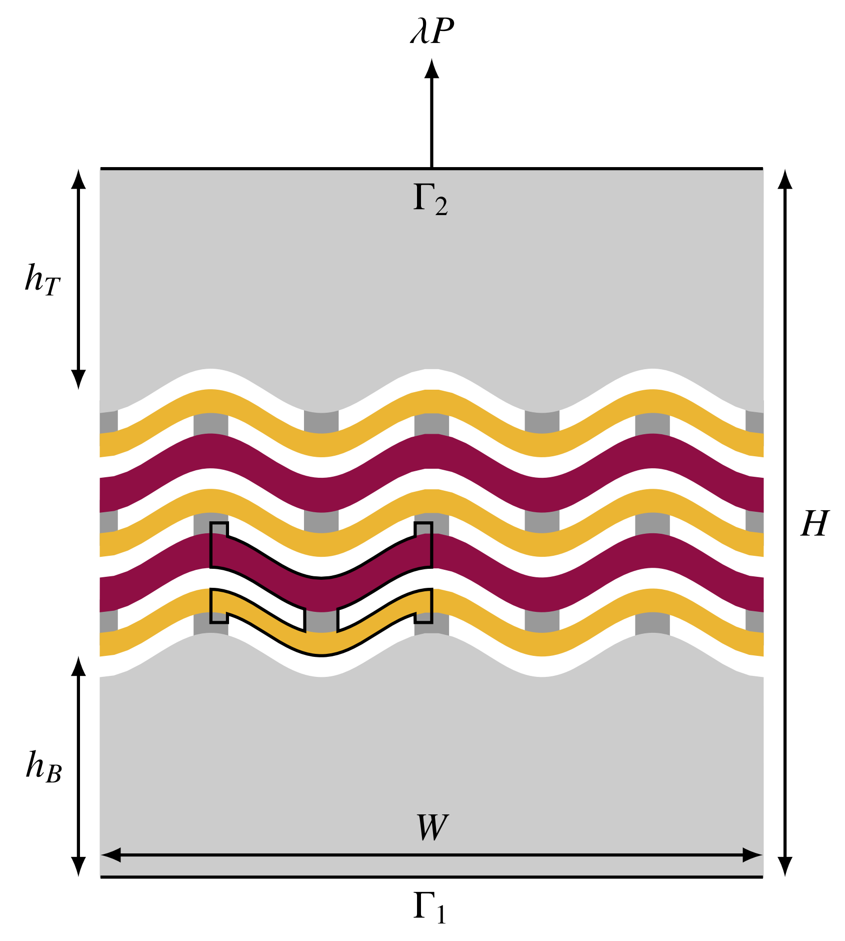
(b) $\Delta L = 2.5$

| Workers | ASPALM | | | APALM |
|---|---|---|---|---|
| # | Serial + | Parallel = | Total | Parallel |
| 0 | 507.2 | 1,778.1 | 2,285.3 | 2,187.1 |
| 1 | 500.5 | 1,757.7 | 2,258.2 | 2,310.2 |
| 2 | 447.5 | 835.3 | 1,282.9 | 1,114.0 |
| 4 | 493.4 | 449.4 | 942.8 | 558.1 |
| 8 | 496.8 | 223.2 | 720.0 | 453.9 |
| 16 | 503.3 | 113.0 | 616.2 | 483.6 |
| 32 | 493.2 | 58.1 | 551.3 | 510.9 |
| 64 | 504.2 | 29.2 | 533.4 | 498.3 |
| 128 | 501.0 | 20.2 | 521.3 | 494.7 |
| 256 | 505.5 | 18.8 | 524.3 | 509.6 |

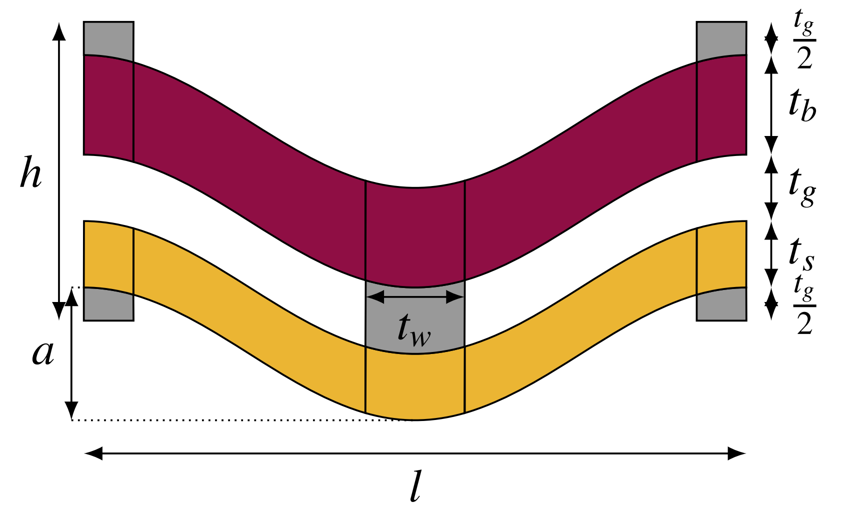
H.M. Verhelst, J.H. Den Besten, and M. Möller (2024) An adaptive parallel arc-length method, Computers & Structures 296:107300

# Example: snapping meta-material

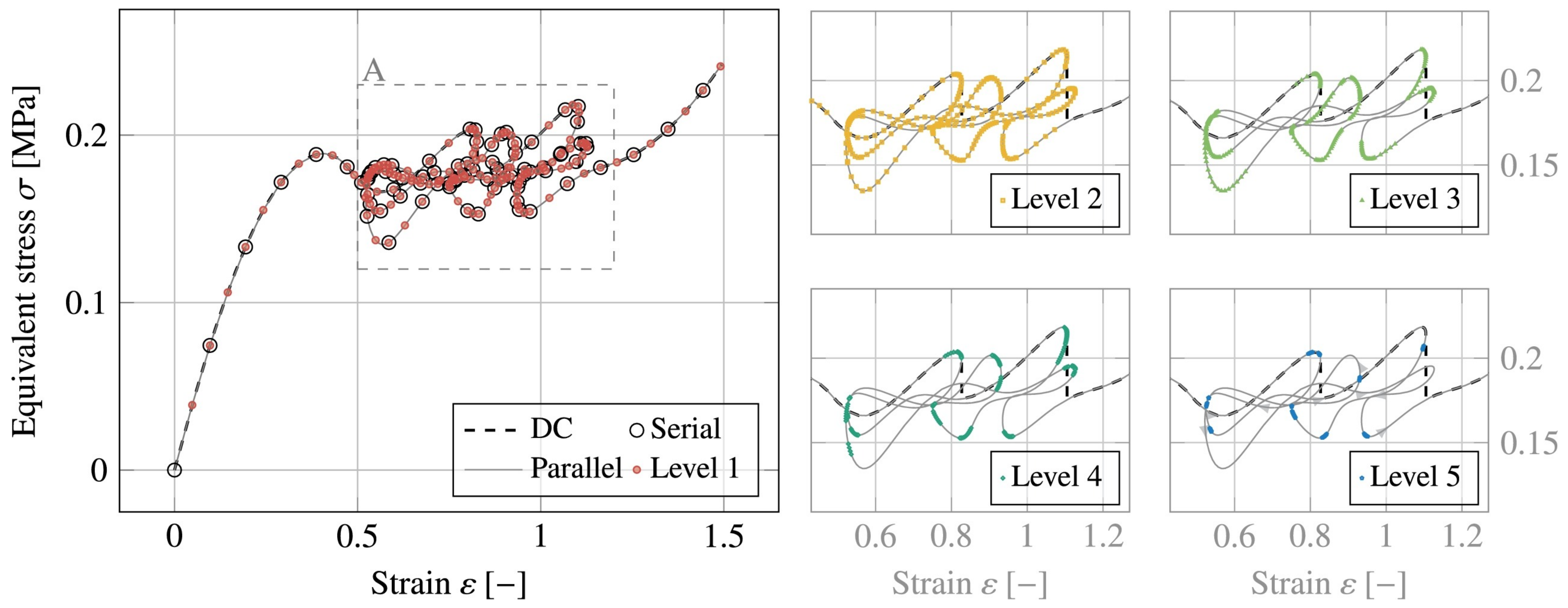
- Compressible Neo-Hookean material model
- 132 B-spline patches and 16.563 dofs in total



(a) A snapping meta-material with $3 \times 2.5$ building blocks, of which one is outlined. The total multi-patch consists of 132 patches.



(b) The snapping building block, composed of 15 patches outlines in black.



H.M. Verhelst, J.H. Den Besten, and M. Möller (2024) An adaptive parallel arc-length method, Computers & Structures 296:107300

## Example: snapping meta-material

- DelftBlue: Intel Xeon Gold  6248R, 24 cores @ 3.0 GHz

| Workers | ASPALM | | | APALM |
|---|---|---|---|---|
| # | Serial + | Parallel = | Total | Parallel |
| 0 | 1,571.6 | 5,204.8 | 6,776.4 | 7,022.9 |
| 1 | 1,686.9 | 4,593.2 | 6,280.1 | 5,319.1 |
| 2 | 1,237.5 | 3,005.9 | 4,243.4 | 3,827.9 |
| 4 | 1,742.7 | 1,548.2 | 3,290.9 | 2,137.3 |
| 8 | 1,445.4 | 717.4 | 2,162.8 | 1,711.8 |
| 16 | 1,931.1 | 352.2 | 2,283.3 | 1,632.9 |
| 32 | 1,746.9 | 219.7 | 1,966.6 | 1,755.6 |

H.M. Verhelst, J.H. Den Besten, and M. Möller (2024) An adaptive parallel arc-length method, Computers & Structures 296:107300

## Lessons learned

- MPI can be used for “task-based” dynamic parallelisation based on a task queue
- The code was parallelised via OpenMP (assembly, solve, etc.) so that the number of MPI processes per compute node was chosen to be smaller (e.g., 1-4) than the total number of compute cores so that each MPI process could solve the problem instance with 6-8 OpenMP threads