

# Non-linear Systems and Parallelization

Gundolf Haase

Karl-Franzens University Graz

Graz, Dec 2025

MONT-BLANC



BioTechMed<sup>®</sup>  
GRAZ

## Bidomain equations with elasticity coupling

The coupling of mechanics with the bidomain is done via additional parameters  $\mathbf{a}$  and  $\mathbf{C}$  which lead to the modified bidomain equations

$$-\nabla \cdot \boldsymbol{\sigma}_e \mathbf{C}^{-1} \nabla \phi_e = \nabla \cdot \boldsymbol{\sigma}_i \mathbf{C}^{-1} \nabla \phi_i + I_e$$

$$\beta I_m = \nabla \cdot \boldsymbol{\sigma}_i \mathbf{C}^{-1} \nabla \phi_i$$

$$I_m = C_m \frac{\partial V_m}{\partial t} + I_{ion}(V_m, \boldsymbol{\eta}, \mathbf{a}) - I_i$$

$$\frac{d\boldsymbol{\eta}}{dt} = f(t, \boldsymbol{\eta}, \mathbf{a})$$

$$V_m = \phi_i - \phi_e$$

The coupling parameter  $\mathbf{a}$  contains quantities calculated from the deformation model [Pathmanathan/Whiteley 2009], with the equilibrium equations of large elastic deformations describing the mechanic behavior,

# Large elastic deformations

$$-\operatorname{div} \sigma(u) = f \quad ,$$

with: stress tensor  $\sigma = 2J^{-1}F \frac{\partial \Psi(C)}{\partial C} F^\top$

deformation gradient tensor  $F_{ij} = \frac{\partial x_i}{\partial X_j}$

compressible  $J = \det F \neq 1$

Cauchy-Green deformation tensor  $C = F^\top F,$

where

$$\Psi = U(J) + \bar{\Psi}_{\text{iso}} + \bar{\Psi}_{\text{aniso}} + \bar{\Psi}_{\text{act}}(\mathbf{a}, \eta(V_m)), \quad U(J) = \frac{\kappa}{2}(J - 1)^2 \quad .$$

is the Helmholtz free energy both the isotropic and anisotropic response, an active part due to the excitation-induced contraction during the course of depolarization.

[Kroon/Holzapfel 2009; Pathmanathan/Whiteley 2009]

# Non-linear elasticity: a sketch

We have to solve the non-linear (quasi-linear) system of equations

$$K(u) \cdot u = f$$

with deformation  $u$  via Newton iteration.

Per Newton step we have to solve

$$A'(u) \cdot \partial u = g(u) \tag{1}$$

with a priori accumulation of

$$A'(u) := \frac{\partial K(u)}{\partial u} \cdot u + K(u) \tag{2}$$

$$g(u) := f - K(u) \cdot u$$

# Non-linear elasticity: algorithmic view I

In each Newton iteration step we have to perform:

- 1 Generate  $A'(u)$ ,  $g(u)$
- 2 Solve (1) with PCG
  - 1 Setup of AMG-preconditioner
    - 1 Find coarse/fine nodes
    - 2 Calculate interpolation
    - 3 Calculate coarse grid operators
  - 2 Apply AMG-PCG
- 3 Update  $\hat{u} = u + \partial u$
- 4 Goto 1

# Non-linear elasticity: algorithmic view I

In each Newton iteration step we have to perform:

- 1 Generate  $A'(u)$ ,  $g(u)$
- 2 Solve (1) with PCG
  - 1 Setup of AMG-preconditioner
    - 1 Find coarse/fine nodes
    - 2 Calculate interpolation
    - 3 Calculate coarse grid operators
  - 2 Apply AMG-PCG
- 3 Update  $\hat{u} = u + \partial u$
- 4 Goto 1

Take into account:

- matrix computation/accumulation might consume **50% of overall time** (expensive operations to determine occupancy pattern of sparse matrices)
- **memory transfer** CPU  $\leftrightarrow$  GPU is expensive

# Non-linear elasticity: algorithmic view I

In each Newton iteration step we have to perform:

- 1 Generate  $A'(u)$ ,  $g(u)$
- 2 Solve (1) with PCG
  - 1 Setup of AMG-preconditioner
    - 1 Find coarse/fine nodes
    - 2 Calculate interpolation
    - 3 Calculate coarse grid operators
  - 2 Apply AMG-PCG
- 3 Update  $\hat{u} = u + \partial u$
- 4 Goto 1

Take into account:

- matrix computation/accumulation might consume **50% of overall time** (expensive operations to determine occupancy pattern of sparse matrices)
- **memory transfer** CPU  $\leftrightarrow$  GPU is expensive

$\implies$  Move more computations and **data** onto the GPU.

# Non-linear elasticity: algorithmic view II

- 1 Setup of operators (once)
  - 1 Calculate sparse pattern of  $K(u)$ ,  $A'(u)$  (→ patterns)
  - 2 Calculate  $K(u)$ ,  $A'(u)$ ,  $g(u)$
  - 3 Setup of AMG-preconditioner wrt.  $A'(u)$  (→ patterns)
- 2 Solve (1) with PCG (each iteration)
  - 1 Setup of AMG-preconditioner wrt.  $A'(u)$   
fixed sparsity pattern of operators
    - 1 Update interpolation
    - 2 Update coarse grid operators
    - 3 no setup
  - 2 Apply AMG-PCG
- 3 Update  $\hat{u} = u + \partial u$
- 4 Update  $K(\hat{u})$ ,  $A'(\hat{u})$ ,  $g(\hat{u})$  (fixed sparsity pattern)
- 5 Goto 2

# Non-linear elasticity: algorithmic view III

## 1 Setup of operators

1 Calculate sparse pattern of  $K(u)$ ,  $A'(u)$

2 Calculate  $K(u)$ ,  $A'(u)$ ,  $g(u)$

GPU: Costa (✓)

3 Setup of AMG-preconditioner wrt.  $A'(u)$  ( $\rightarrow$  patterns)

## 2 Solve (1) with PCG

1 Setup of AMG-preconditioner wrt.  $A'(u)$   
fixed sparsity pattern of operators

1 Update interpolation

GPU: Neic ✓

2 Update coarse grid operators

GPU: Neic ✓

3 no setup

2 Apply AMG-PCG

GPU: done ✓

3 Update  $\hat{u} = u + \partial u$

GPU: Costa

4 Update  $K(\hat{u})$ ,  $A'(\hat{u})$ ,  $g(\hat{u})$  (fixed sparsity pattern)

GPU: Costa (✓)

5 Goto 2

## Local calculation of stiffness matrix (potential)

- Xeon E5645 with 6 cores; 2.4 GHz, 12MB of cache; 32 GB/sec bandwidth
- GTX 680 with 1536 CUDA cores, 1006 MHz Base Clock; 192.2 GB/sec memory bandwidth.
- `nvcc -O3 ; gcc -O3`
- tetrahedrons with linear test functions

n elements	Std. vs. Vectorized	Vectorized vs. CUDA	Std. vs. CUDA
12.500	6.3	34	214
50.000	5.3	67	353
112.500	5.4	76	410
450.000	4.9	<b>84</b>	<b>416</b>
1.250.000	5.5	74	408

Table: Speedup (wrt. one CPU-core) of calculating the local stiffness matrices.

# Global assembling of stiffness matrix on CPU/GPU

Assume: **known pattern** of global stiffness matrix (CSR; done in setup once)

Two approaches:

- 1 store all element matrix entries globally into **one large array** (coord);
  - ▶ perfect parallel computation (vectorized)
  - ▶ precomputed accumulation pattern (from setup)
  - ▶ global accumulation is perfectly parallel
  - ? **memory** transfer Liebmann/Neic [LN]
- 2 store local matrices only temporarily (or not at all); **accumulate directly** into global matrix
  - **offset** information per element needed [DUNE also]
  - ▶ saves memory accesses
  - !! resource conflicts when global matrix entries are updated  
⇒ **coloring of elements** Haase/Hraßnigg [HH]

# Matrix assembling - OpenMP: May 9, 2014

Timing in sec.; 2 Mill. tetrahedra, linear test functions, potential problem  
Workstation with Xeon E5-2600 v2, 10 cores

cores	LN	HH coloring	HH reord.	HH atomic +=	HH LN-array
1	0.88+0.51	1.80	1.06	<b>1.04</b>	0.82+0.51
2	0.80+0.26	1.08	<b>1.01</b>	1.04	0.75+0.26
4	0.70+0.22	0.67	0.41	<b>0.41</b>	0.36+0.22
10	0.68+0.10	0.53	<b>0.34</b>	0.39	0.28+0.10
20	0.54+0.05	0.36	0.33	<b>0.27</b>	0.23+0.05

Conclusion: **Atomic** operations are so **fast**, and potential conflicts so less, that **coloring doesn't pay off**  
— at least for the test environment.

## Matrix assembling - OpenMP: July 7, 2014

Timing in sec.; 15 Mill. tetrahedra, linear test functions, potential problem  
Workstation with Xeon E5-2600 v2, 10 cores

cores	LN	HH coloring	HH atomic +=
1	7.3	<b>3.9</b>	5.5
2	4.6	<b>2.9</b>	3.4
4	3.6	2.4	<b>2.4</b>
8	2.8	2.4	<b>2.2</b>
10	3.3	2.3	<b>2.2</b>
20	4.9	1.8	<b>1.7</b>

Conclusion: **Atomic** operations are so **fast**, and potential conflicts so less, that **coloring doesn't pay off**

Implementation LN suffers much more from bandwidth saturation.  
HH (atomic) is still an older code version.

## Matrix assembling - OpenACC [since PGI 15.10]

```
#pragma acc kernels pcopyout(sk[0:nnz], f[0:nnode]),  
                    pcopyin(ia[0:ndof_e*nelem], xc[0:2*nnode], id[0:nnode+1], ik[0:nnz])  
{  
    const int idn = id[nnode];  
#pragma acc loop  
    //for (int k = 0; k < id[nnode]; ++k) { // will not be parallelized  
    for (int k = 0; k < idn; ++k) { // but this will parallelize  
        sk[k] = 0.0; }  
#pragma acc loop  
    for (int k = 0; k < nnode; ++k) {  
        f[k] = 0.0; }  
  
    float ske[3][3], fe[3];  
#pragma acc loop private(ske, fe) vector(32)  
    for (int i = 0; i < nelem; ++i) {  
        CalcElem(ia + 3 * i, xc, ske, fe);  
        AddElem(ia+3*i, ske, fe, id, ik, sk, f);  
    }  
    return;  
}
```

## Matrix assembling - OpenACC cnt.

```
#pragma acc routine seq
void CalcElem(const int ial[3], const float xc[], float ske[3][3], float fe[3])
{
    // sequ. code for one element
}

#pragma acc routine seq
void AddElem(const int ial[3], const float ske[3][3], const float fe[3],
             const int id[], const int ik[], float sk[], float f[])
{
    #pragma acc loop // needed [PGI 15.10]
        for (int i = 0; i < 3; ++i) {
        const int ii = ial[i];
    #pragma acc loop // needed [PGI 15.10]
        for (int j = 0; j < 3; ++j) { // no symmetry
            const int jj = ial[j];
            const int ip = fetch(ii, jj, id, ik);
    #pragma acc atomic update // no atomic possible in seq-routine before PGI 15.5
            sk[ip] += ske[i][j];
        }
    #pragma acc atomic update // no atomic possible in seq-routine before PGI 15.5
            f[ii] += fe[i];
        }
}
```

## What about Xeon Phi?

Xeon Phi 60 cores (1GHz), 8 GB vs. 2×Xeon E5-26508 (2GHz), 2×8 Cores

No MIC pragmas needed: OpenMP 4.0 fully in Intel compiler included

- Quantum mechanics,  $1024 \times 1024$  points, 6 dimensions
- OpenMP on Host as well as on MIC (**native** mode)
- AVX and AVX2 vectorization in work for QR factorization.
- code by M. Liebmann / D. Sattlegger / M. Alinejadmofrad [May 9, 2014]

cores	Host		MIC			
	1	16	60	120	180	240
time(sec.)	41.4	<b>2.66</b>	6.6	3.8	2.9	<b>2.4</b>
efficiency	1.00	0.97	1.00	0.87	0.76	0.70

In progress

- **native** mode: all code and all data on MIC
- **offload** mode: explicit data transfer to/from MIC

# Linear elasticity

Lamé equations: linear elasticity for small deformations  $\left| \frac{\partial u_l}{\partial x_k} \right| \ll 1$

$$-\operatorname{div} \sigma(u) = f \quad ,$$

with: stress tensor  $\sigma = D\varepsilon := 2\delta_{i,j}\varepsilon_{i,i} + 2\mu\varepsilon_{i,j}$

Cauchy deformation tensor:  $\varepsilon_{k,l} = \frac{1}{2} \left( \frac{\partial u_l}{\partial x_k} + \frac{\partial u_k}{\partial x_l} \right)$

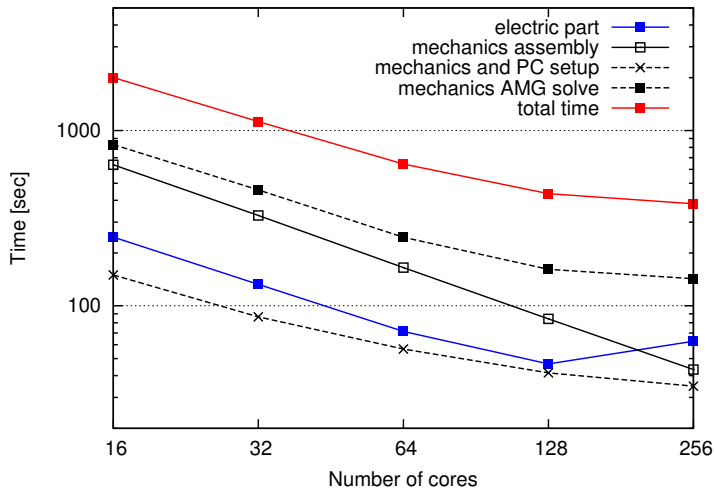
Tetrahedra with linear f.e. functions.

# AMG components

- Coarsening by agglomeration regarding strong connections.
- Intergrid transfer by constant interpolation/averaging
  - ▶ matrix dependent intergrid transfer: less iterations but slower in sum
- Block ( $3 \times 3$ ) Jacobi-smoother
- still the older communicator
- still the old coarse grid parallelization

## Example: elasticity + potential problem

TBunnyC: 862,515 vertices ( $n + 3 \times n$  dofs) on CPU [GPU: 10 $\times$ ]



©C.Augustin