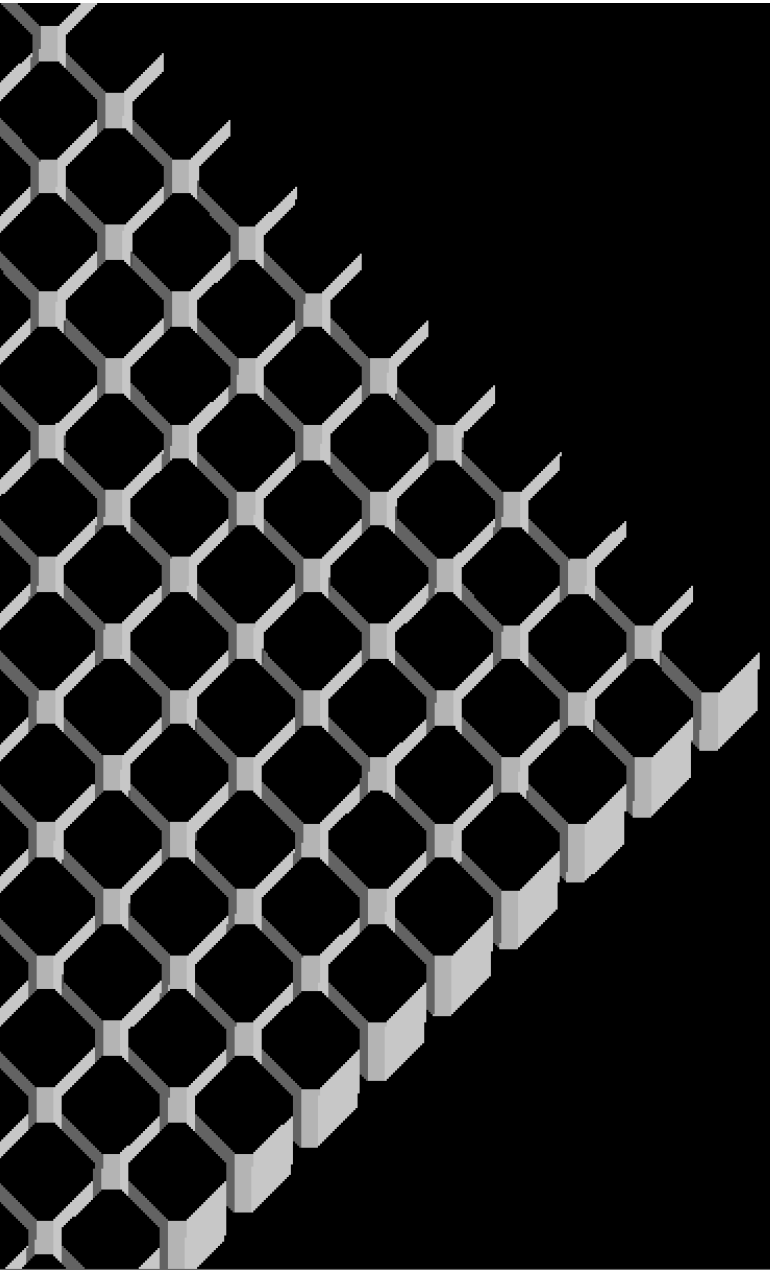


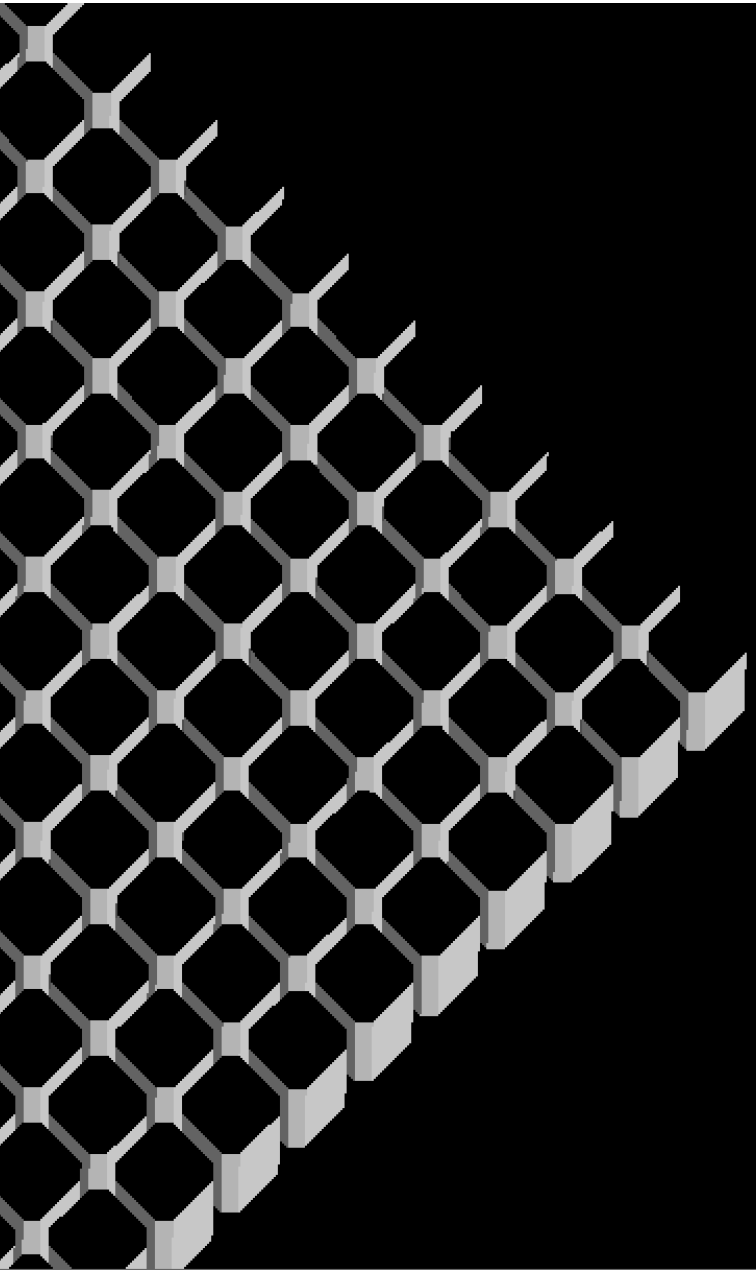
# *High Performance Computational Mechanics*

# I. Concepts

1. Context
2. What to parallelize
3. Read and partition mesh
4. Assembly
5. Solver
6. Basic operations
   - Matrix-vector product
   - Scalar product

# II. Implementation

1. MPI point-to-point
2. OpenMP
3. Hybrid MPI/OpenMP
4. Dynamic load balance
5. Performance analysis
6. What else
   - Pipelined Conjugate Gradient
   - Deflated Conjugate Gradient
   - Restricted Additive Schwarz
   - Mesh Multiplication
7. Future

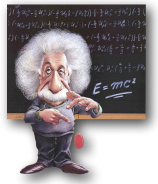


# Context

# High Performance Computational Mechanics

**Physical phenomenon**

Modelling

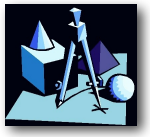



**Partial Differential Equation**

$$\rho\frac{\partial \boldsymbol{u}}{\partial t} + \rho(\boldsymbol{u}\cdot\nabla)\boldsymbol{u} - \nabla\cdot[2\mu\boldsymbol{\varepsilon}(\boldsymbol{u})] + \nabla p = \rho\boldsymbol{f}$$

$$\nabla\cdot\boldsymbol{u} = 0$$

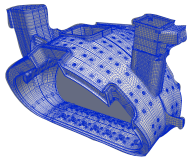
Variational method



**Weak Form**

$$(\boldsymbol{M}\partial_t\boldsymbol{U},\boldsymbol{V}) + B(\boldsymbol{U},\boldsymbol{V}) = L(\boldsymbol{V})$$

Space discretization (mesh)

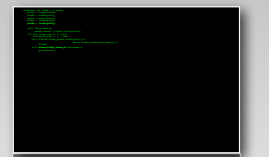


**Algebraic system**

$$\begin{bmatrix} \mathbf{A}_{uu} & \mathbf{A}_{up} \\ \mathbf{A}_{pu} & \mathbf{A}_{pp} \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ \mathbf{p} \end{bmatrix} = \begin{bmatrix} \mathbf{b}_u \\ \mathbf{b}_p \end{bmatrix}$$

Programming

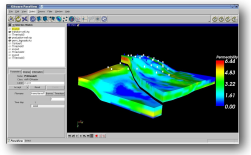
**Parallel code**



**Terabytes**

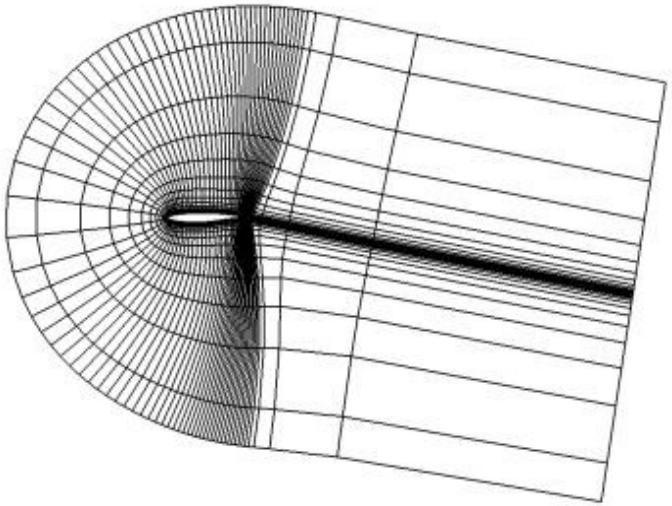
Execution

Visualization



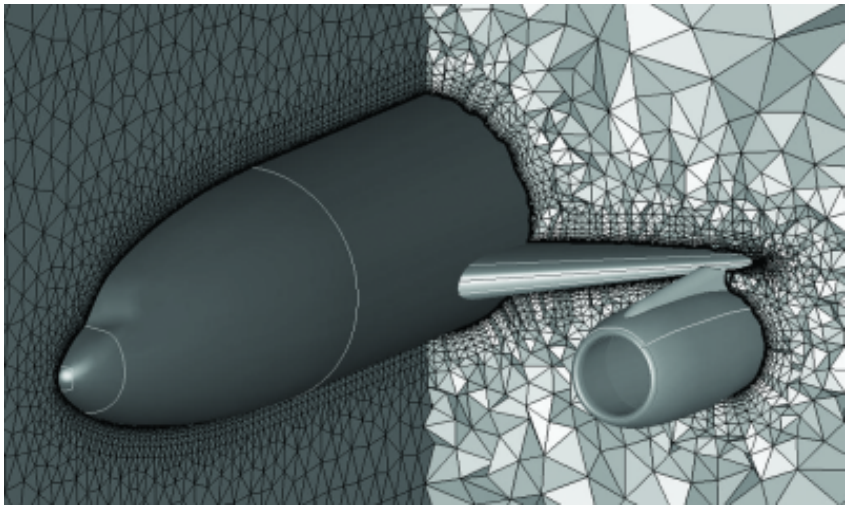
# Structured *vs* unstructured mesh

*Structured*



- Same number of neighboring nodes
- Predictable connectivity
- Finite Difference (FD)

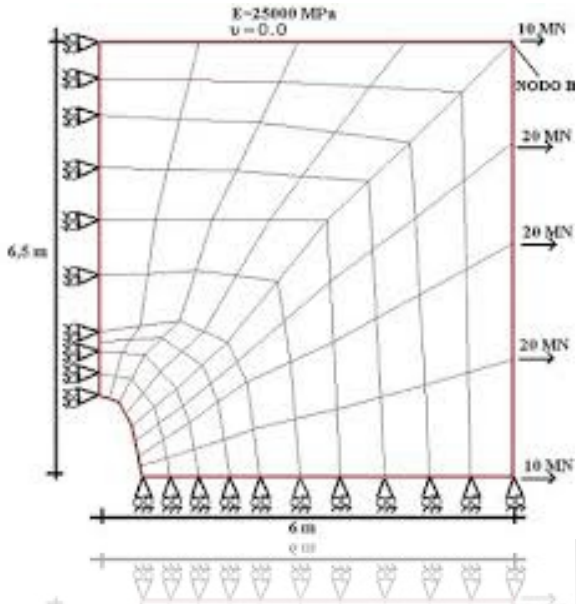
***Unstructured***



- Different numbers of neighboring nodes
- Unpredictable connectivity
- Finite Element (FE), Finite Volume (FV)

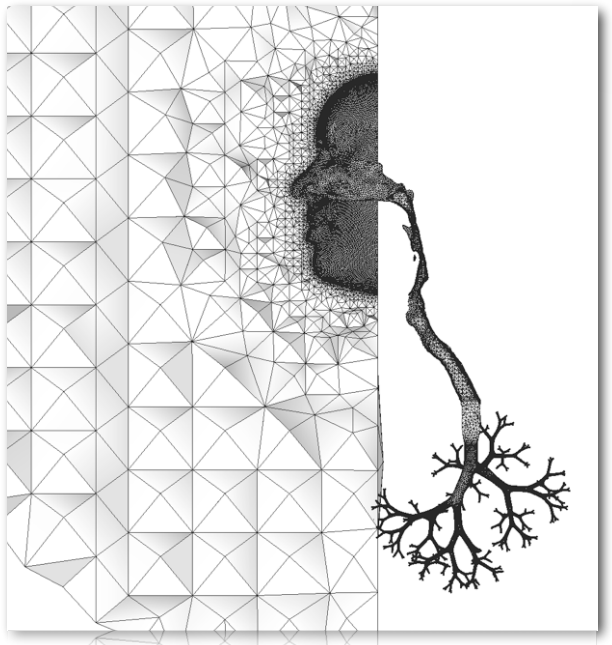
# Small *vs* large problems

*Small*




- Direct solver
- Low memory requirement
- "Easy" algebraic system

***Large***

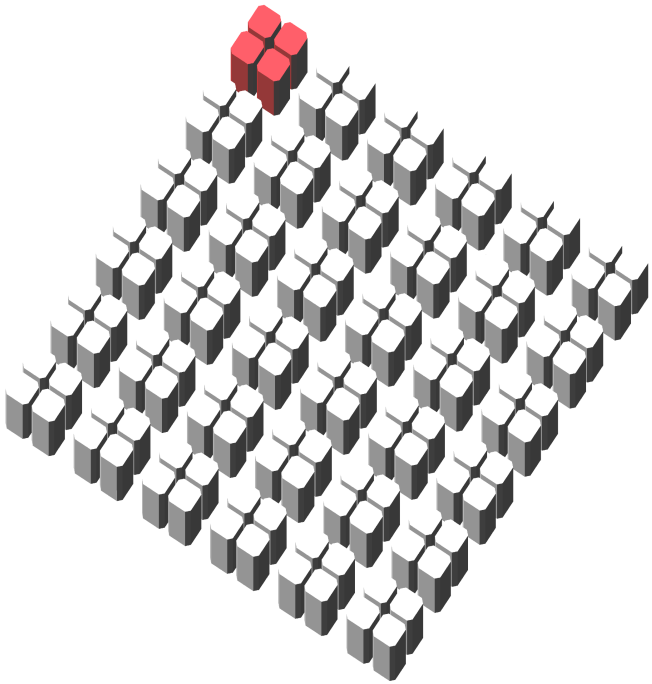


- Iterative solver
- High memory requirement
- Stiff algebraic system

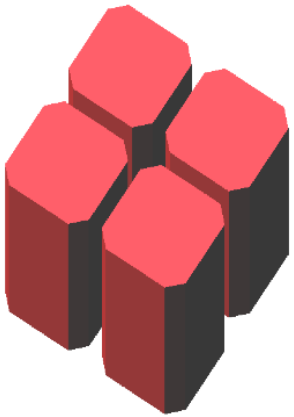
# Why MPI?

MPI, OpenMP or MPI/OpenMP or MPI/OpenMP with accelerators

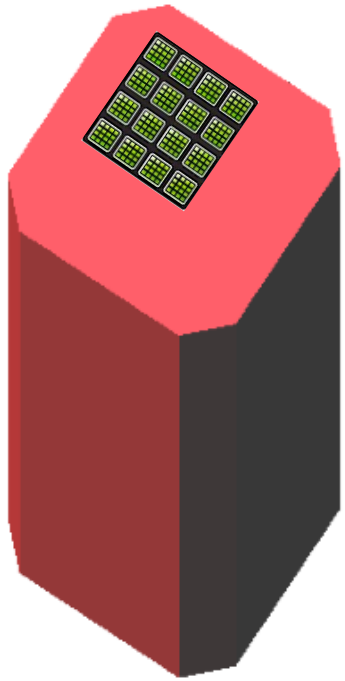
MPI
(between nodes)



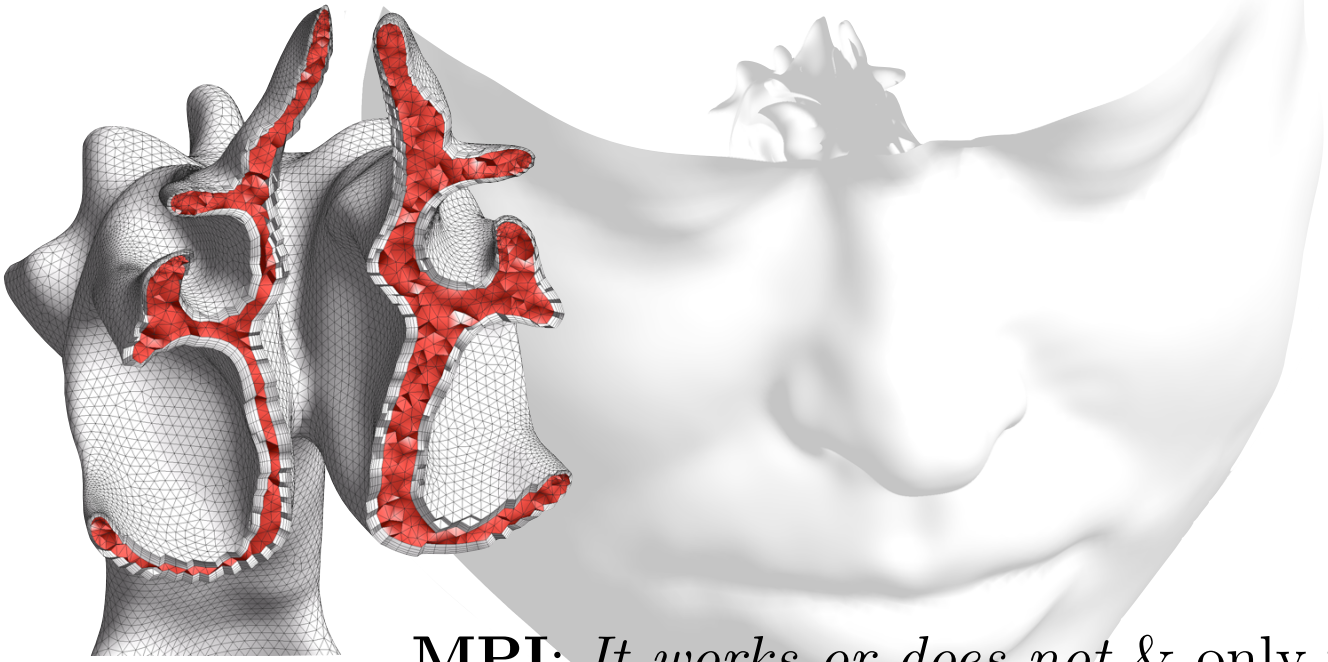
OpenMP
(between cores)



CUDA, etc.
(accelerate cores)



# Why MPI?

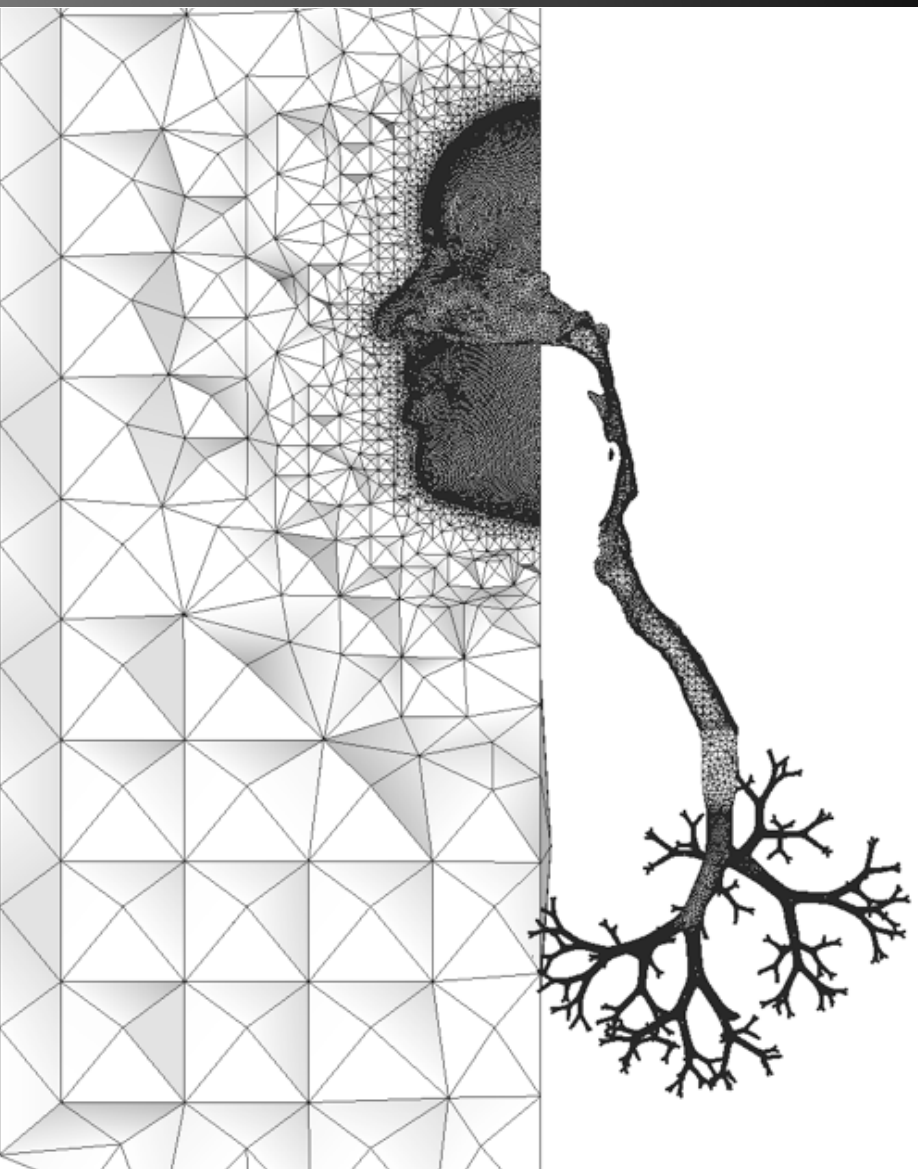


**MPI**: *It works or does not* & only way to go to massively parallel

**OpenMP**: Difficult to maintain

**Accelerators**: Not well suited for unstructured meshes

# Keywords



Unstructured meshes

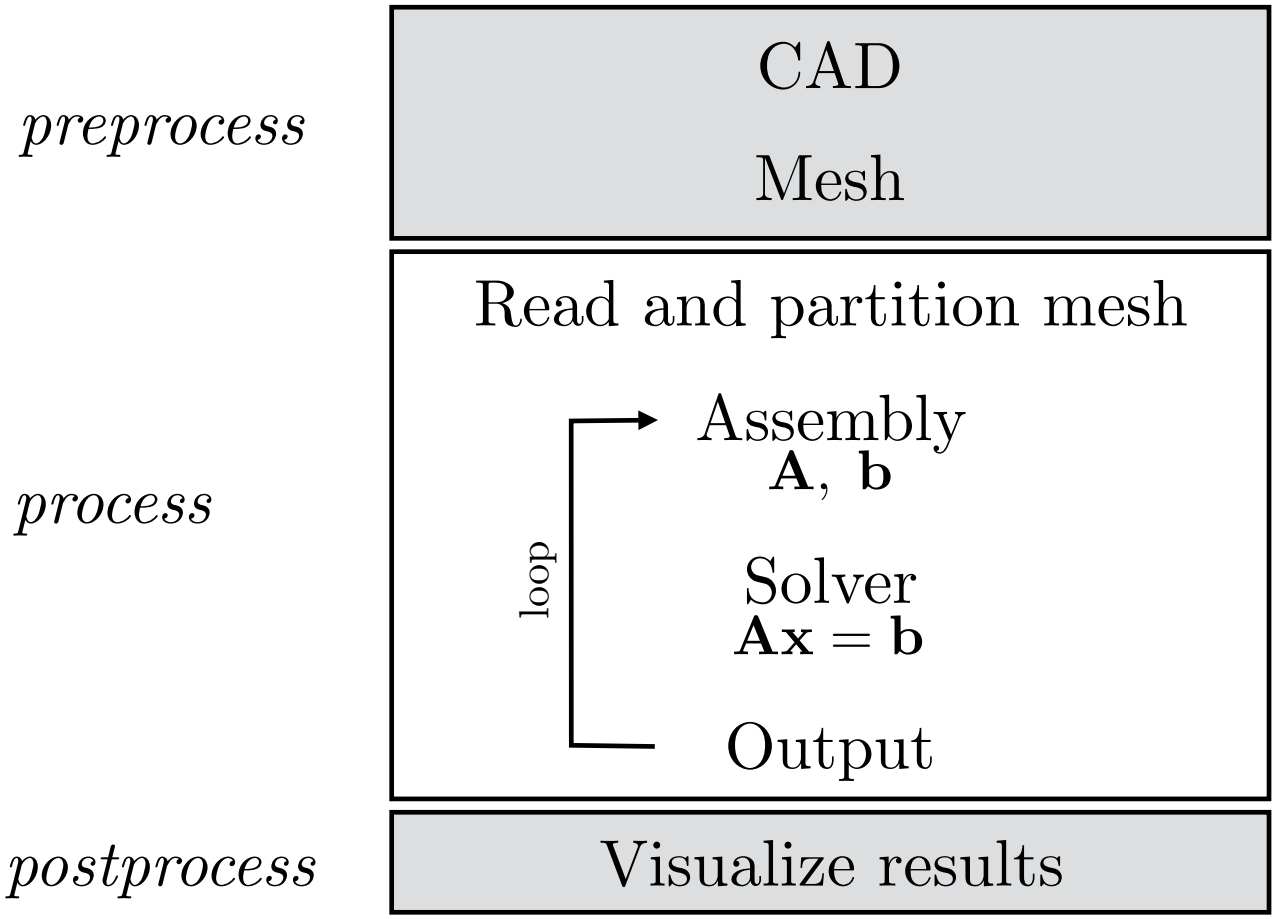
Finite element method

Iterative solvers

MPI, OpenMP

# What to parallelize

# A simulation



# What to parallelize

Parallelize as much as possible

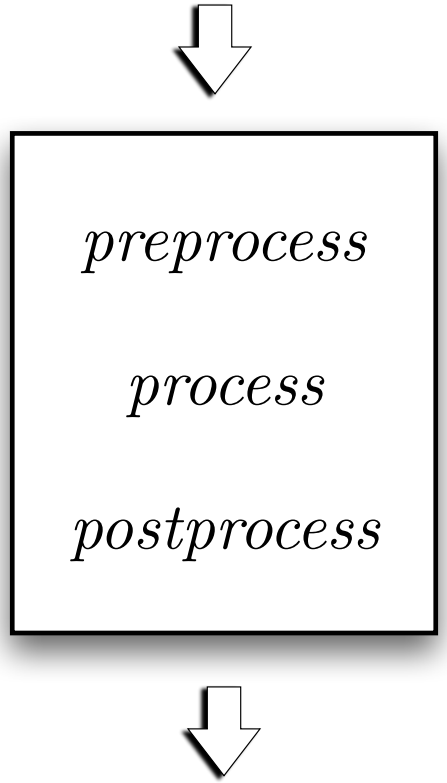
*from*..... *preprocess*

*through*..... *process*

*until*..... *postprocess*

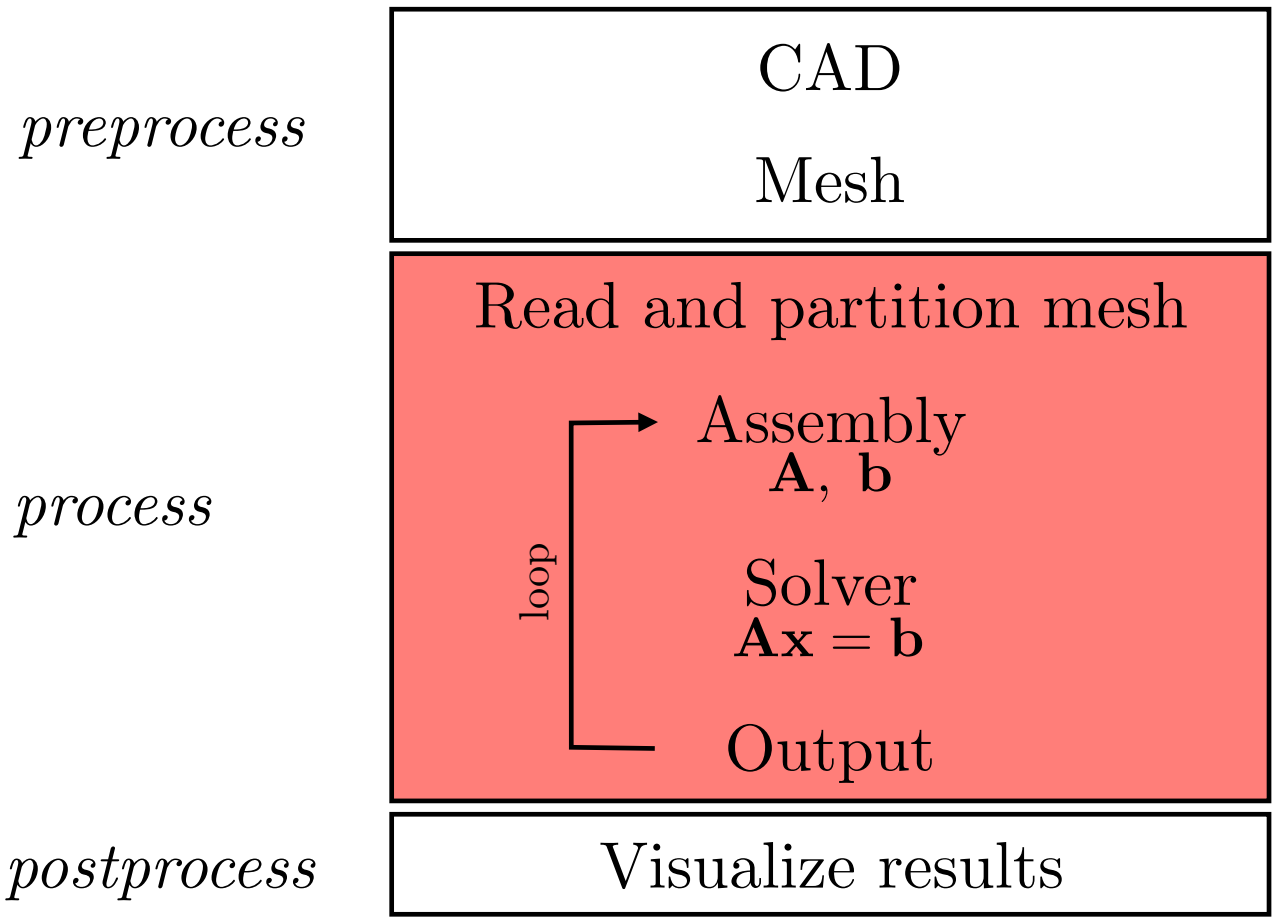
# Ideal situation

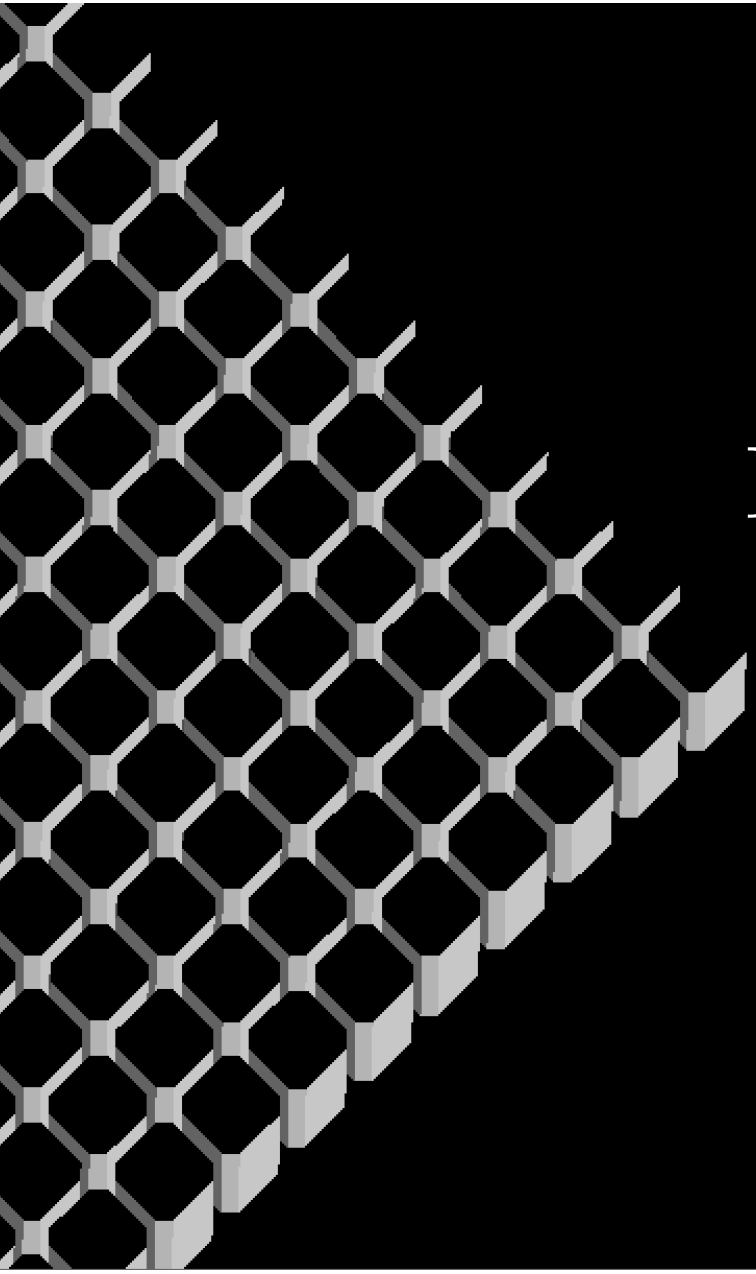
Enter MPI Environment



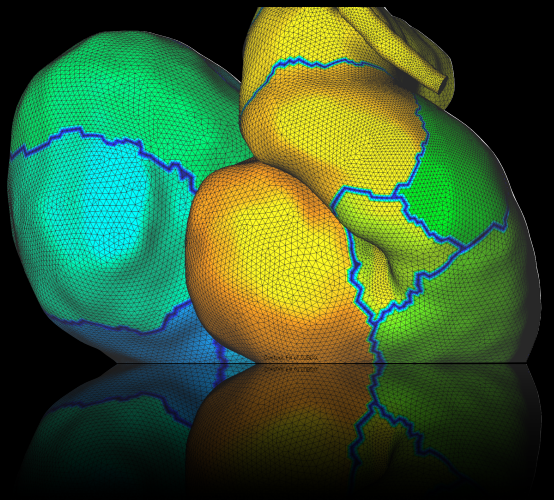

Exit MPI Environment

# What we talk about today



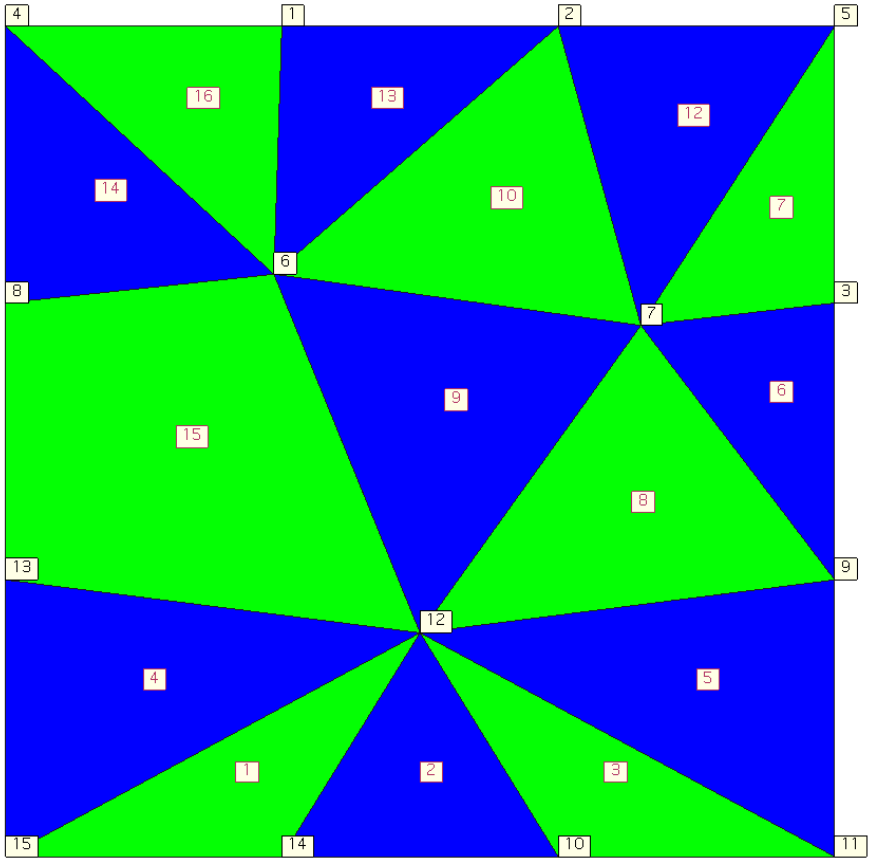


# Read and partition mesh



# Read and partition mesh

## Description of an unstructured mesh

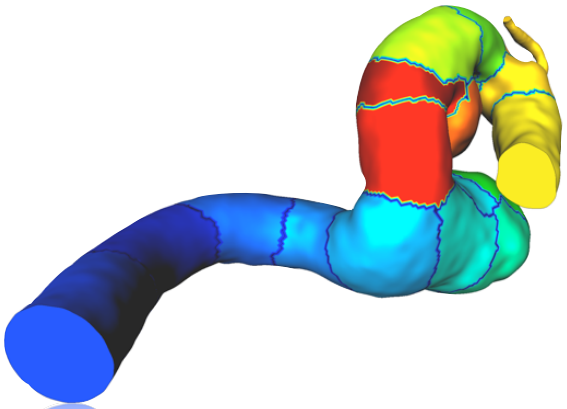



Mesh info

```
elements
 1   15  14  12
 2   14  10  12
 3   12  10  11
 4   15  12  13
 5   12  11   9
 6    9   3   7
 7    7   3   5
 8    9   7  12
 9   12   7   6
10    6   7   2
11    7   5   2
12    2   1   6
13    6   4   8
14   13  12   6   8
15    6   1   4
end elements
```

```
coordinates
1   0.33 1.00
2   0.67 1.00
3   1.00 0.67
4   0.00 1.00
5   1.00 1.00
6   0.32 0.70
7   0.77 0.64
8   0.00 0.67
9   1.00 0.33
10  0.67 0.00
11  1.00 0.00
12  0.50 0.27
13  0.00 0.33
14  0.33 0.00
15  0.00 0.00
16  0.00 0.50
end coordinates
```

# Read and partition mesh



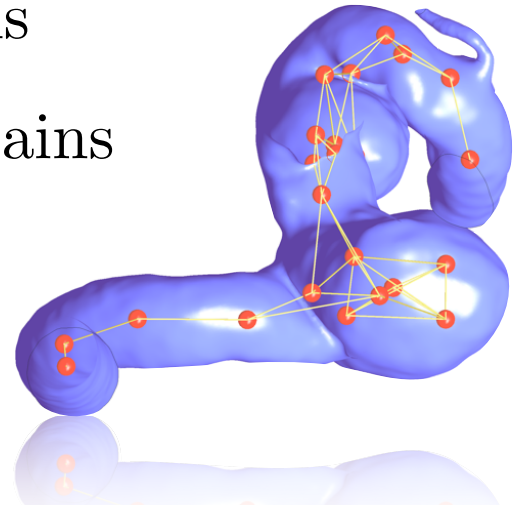
*Mesh partitioning:*

Assign a subdomain to

the elements of the mesh

under the constraints

1. Balance the work between subdomains
2. Minimize the interfaces between subdomains

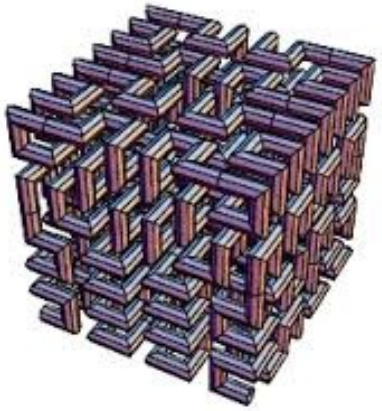


E.g.: METIS (USA), SCOTCH (F), ZOLTAN(USA)

# Read and partition mesh

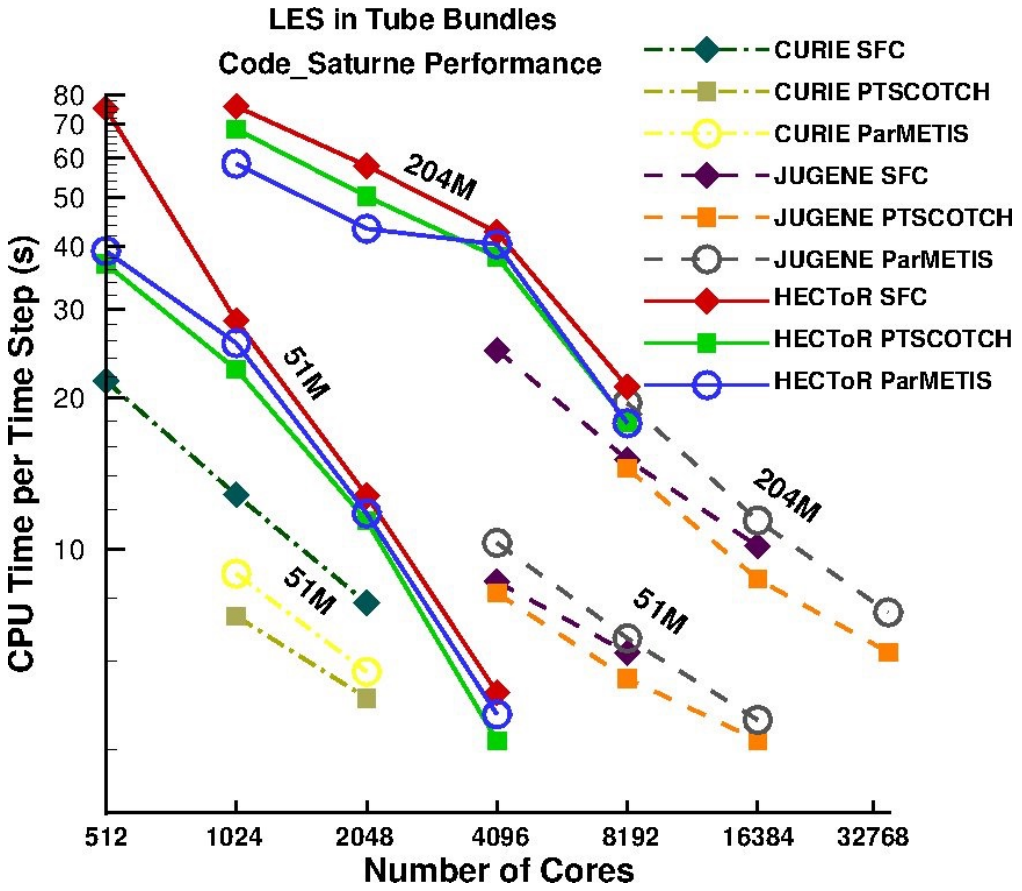
In general, only the *element connectivity* is required to partition the mesh

Counterexample: partitioning based on Space Filling Curve (SFC)



```
elements
 1   15 14 12
 2   14 10 12
 3   12 10 11
 4   15 12 13
 5   12 11  9
 6    9  3  7
 7    7  3  5
 8    9  7 12
 9   12  7  6
10    6  7  2
11    7  5  2
12    2  1  6
13    6  4  8
14   13 12  6  8
15    6  1  4
end elements
```

# Examples




Courtesy of Charles Moulinec (STFC, UK)

# Read and partition mesh

Two possible paradigms

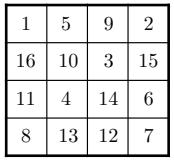
Master - Workers

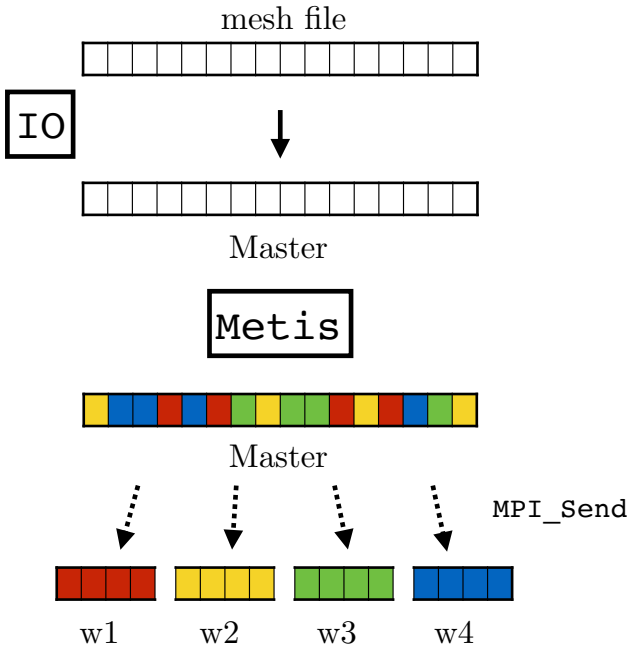
&

Workers only

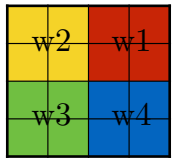
# Read and partition mesh

## Option 1
(sequential)

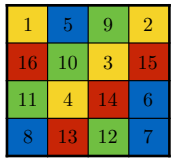


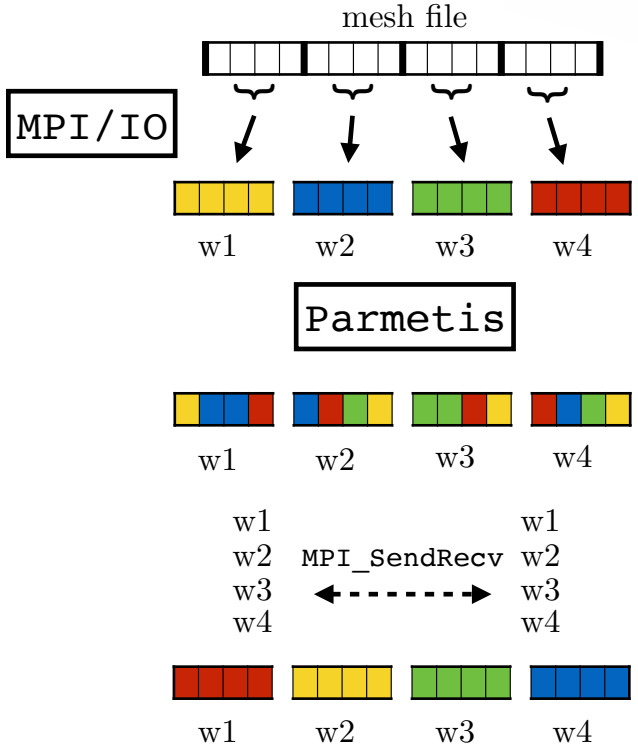




```
elements
 1   15 14 12
 2   14 10 12
 3   12 10 11
 4   15 12 13
 5   12 11  9
 6    9  3  7
 7    7  3  5
 8    9  7 12
 9   12  7  6
10    6  7  2
11    7  5  2
12    2  1  6
13    6  4  8
14   13 12  6  8
15    6  1  4
end elements
```



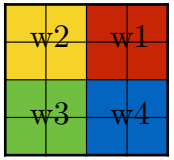

## Option 2
(parallel)



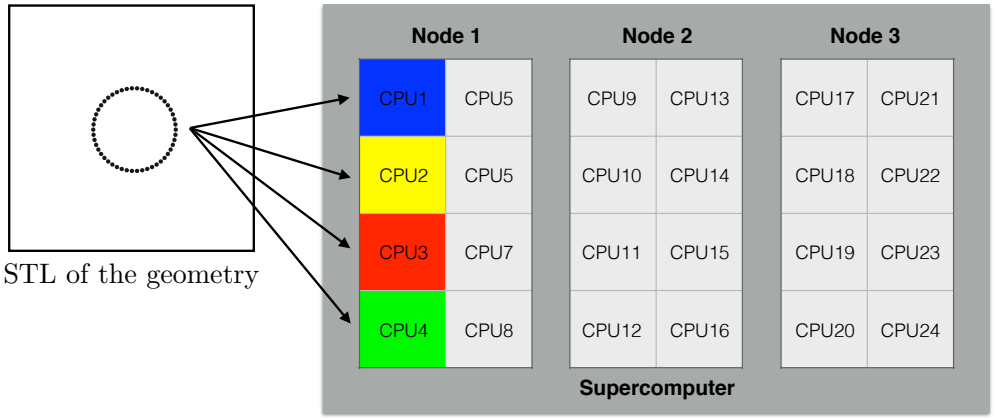
# Read and partition mesh

After partition
workers have their local meshes



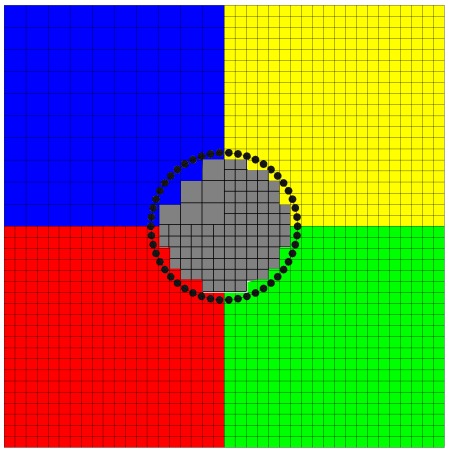

The mesh info of workers depends on the method
FD, FE, FV

# Ideal situation

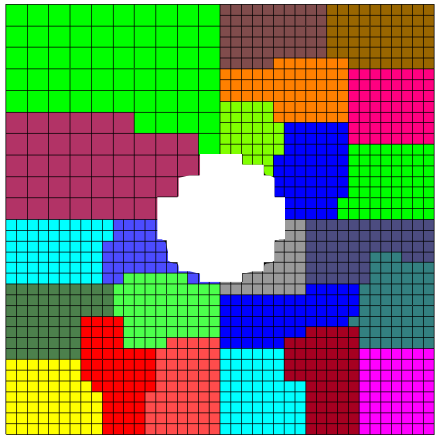



STL of the geometry

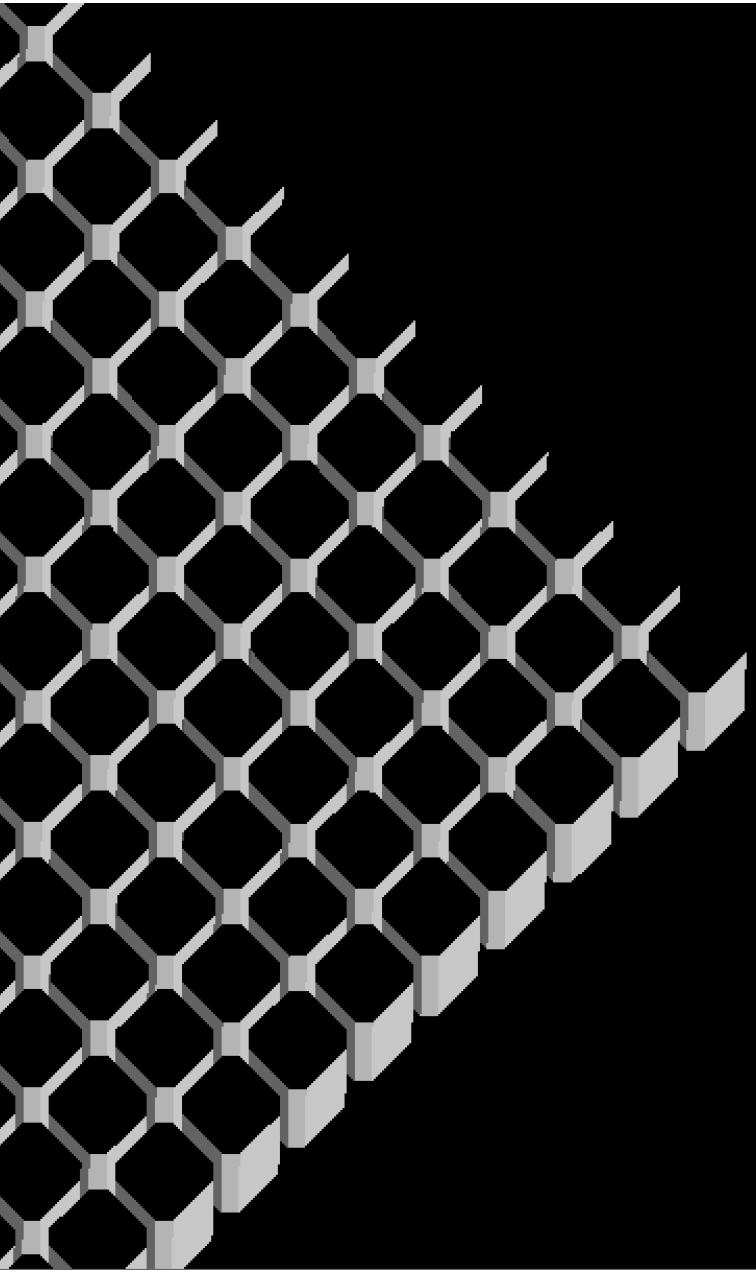
Mesh in parallel on 4 CPUs



Redistribute in parallel the mesh among the 24 CPUs



Run the simulation on 24 CPUs



Assembly

$\mathbf{A},\ \mathbf{b}$

# From equation to assembly

Partial Differential Equation

$$\rho\frac{\partial \boldsymbol{u}}{\partial t} + \rho(\boldsymbol{u}\cdot\nabla)\boldsymbol{u} - \nabla\cdot[2\mu\boldsymbol{\varepsilon}(\boldsymbol{u})] + \nabla p = \rho\boldsymbol{f}$$

$$\nabla\cdot\boldsymbol{u} = 0$$

Discretization method

Finite Difference (FD)

Finite Element (FE)

Finite Volume (FV)

Loop over nodes

Loop over elements

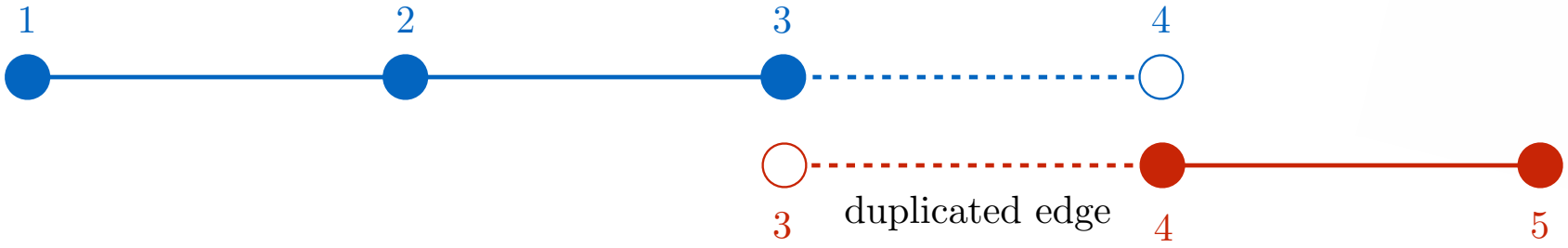
Loop over element faces

Algebraic system

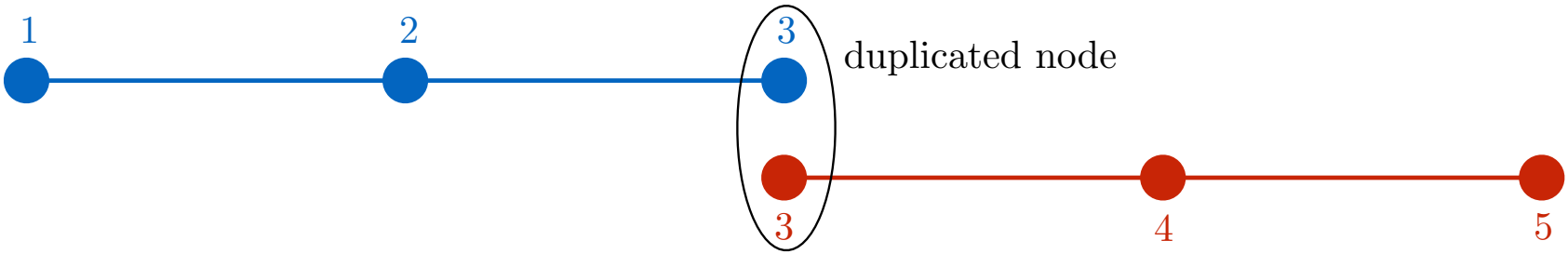
$$\begin{bmatrix} \mathbf{A}_{uu} & \mathbf{A}_{up} \\ \mathbf{A}_{pu} & \mathbf{A}_{pp} \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ \mathbf{p} \end{bmatrix} = \begin{bmatrix} \mathbf{b}_u \\ \mathbf{b}_p \end{bmatrix}$$

# Assembly

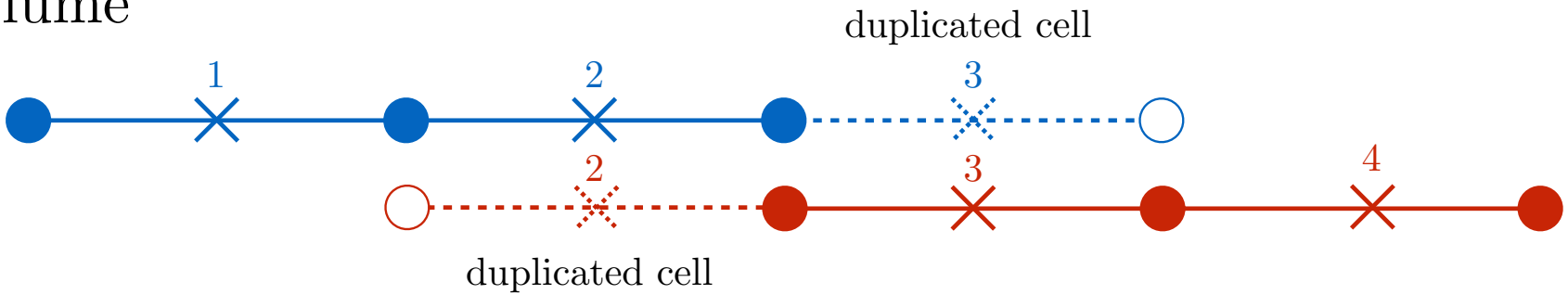
## Finite difference




## Finite element

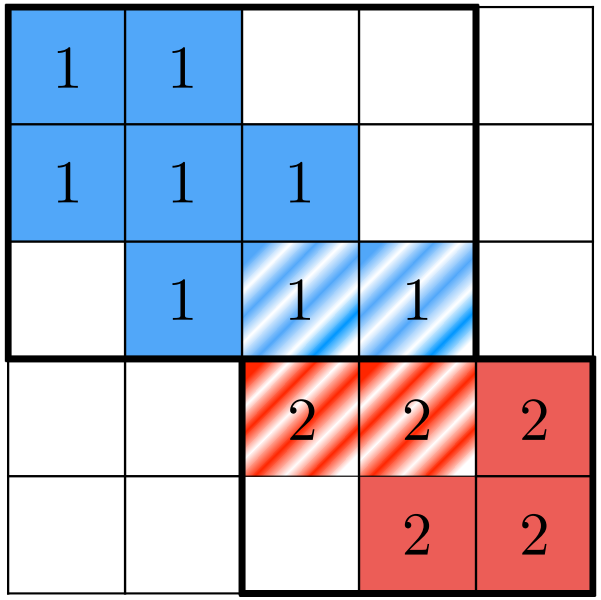


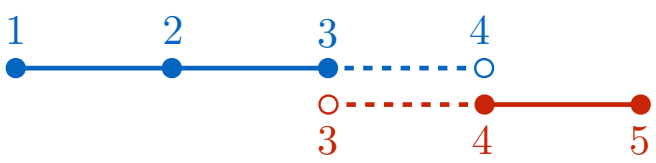

## Finite volume



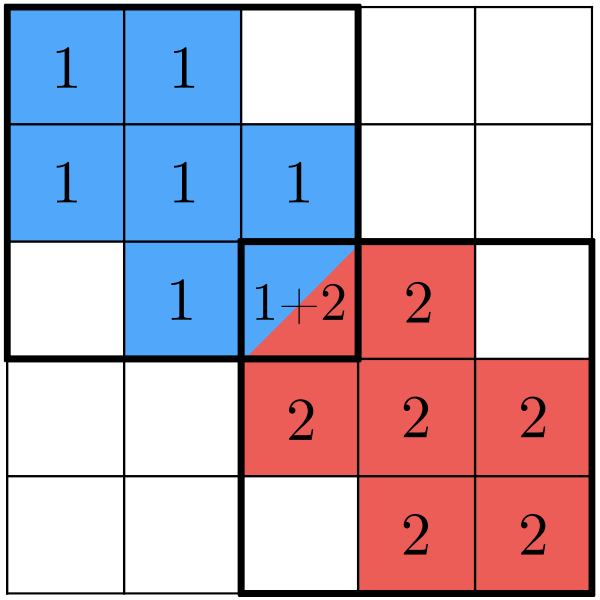
# Assembly

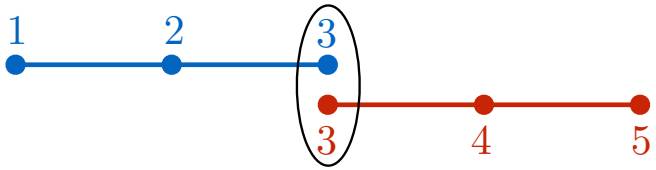
## Finite difference



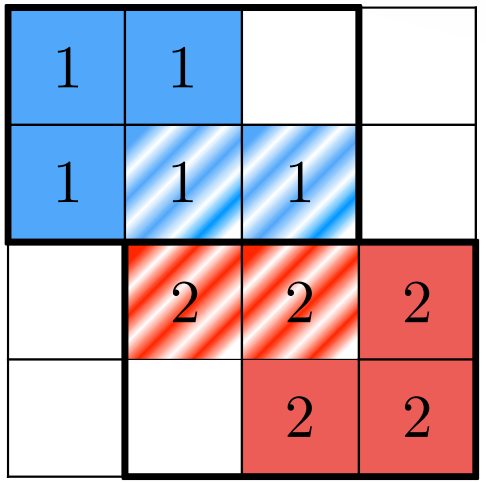



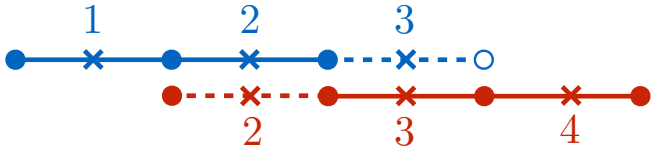

## Finite element



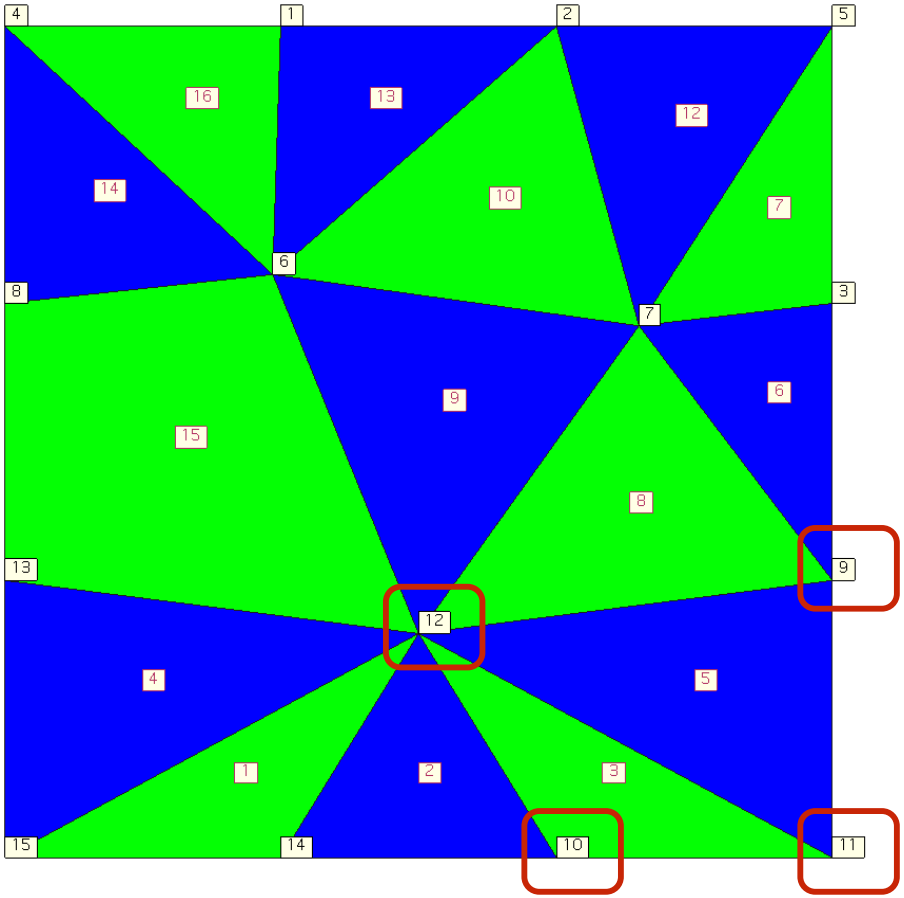




## Finite volume

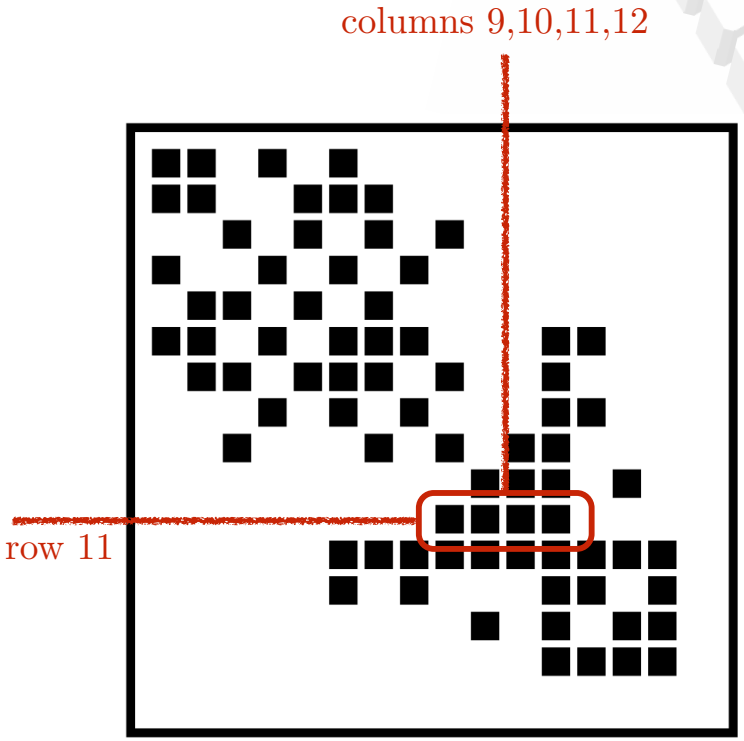


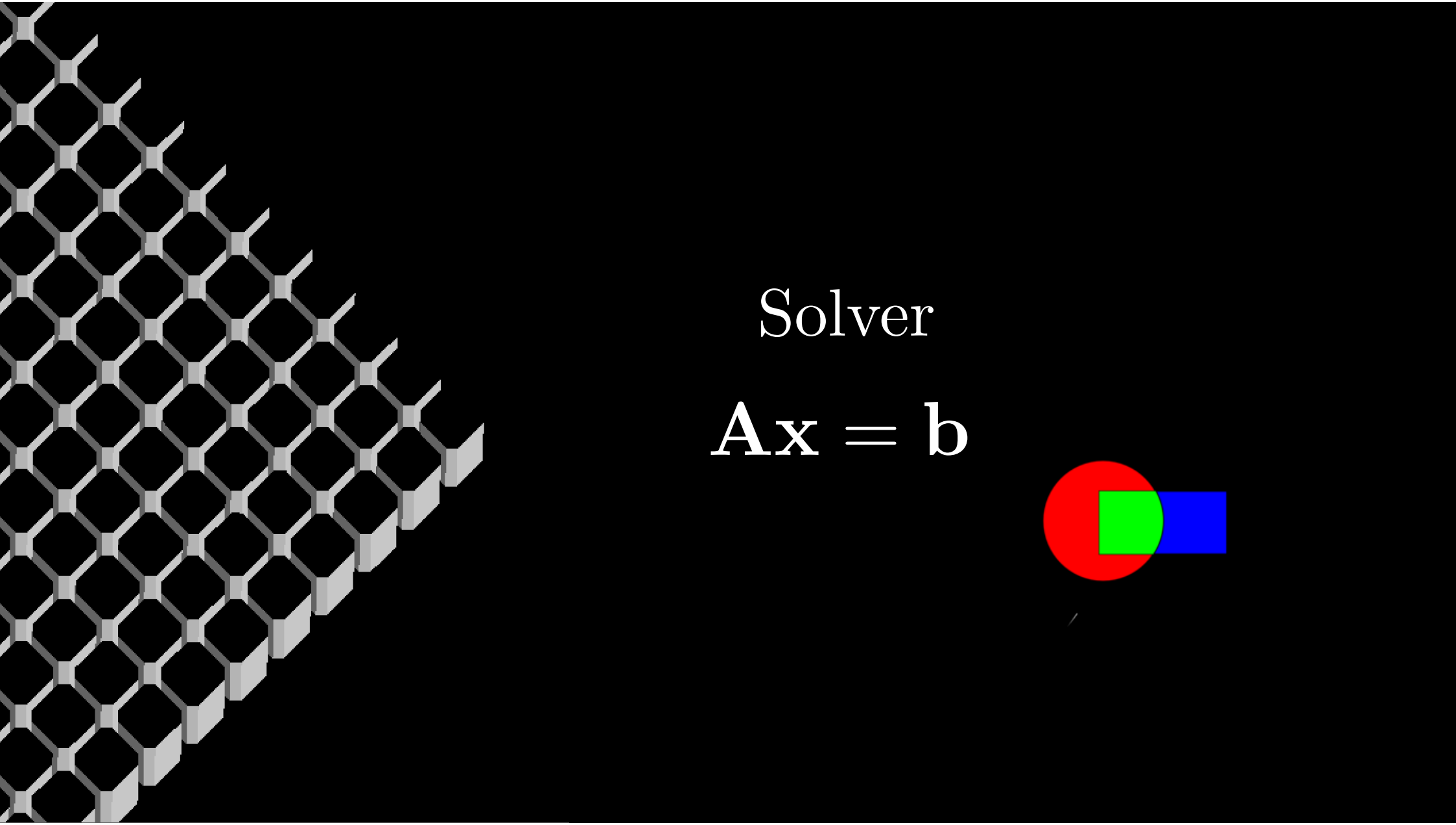



# Sparse matrix: CSR format




```
elements
 1   15  14  12
 2   14  10  12
 3   12  10  11
 4   15  12  13
 5   12  11   9
 6    9   3   7
 7    7   3   5
 8    9   7  12
 9   12   7   6
10    6   7   2
11    7   5   2
12    2   1   6
13    6   4   8
14   13  12   6  8
15    6   1   4
end elements
```




Solver
$\mathbf{Ax} = \mathbf{b}$

# From algebraic system to solution

Algebraic system

$$\begin{bmatrix} \mathbf{A}_{uu} & \mathbf{A}_{up} \\ \mathbf{A}_{pu} & \mathbf{A}_{pp} \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ \mathbf{p} \end{bmatrix} = \begin{bmatrix} \mathbf{b}_u \\ \mathbf{b}_p \end{bmatrix}$$

Iterative solver

$$\begin{bmatrix} \tilde{\mathbf{u}} \\ \tilde{\mathbf{p}} \end{bmatrix} \approx \begin{bmatrix} \mathbf{u} \\ \mathbf{p} \end{bmatrix}$$

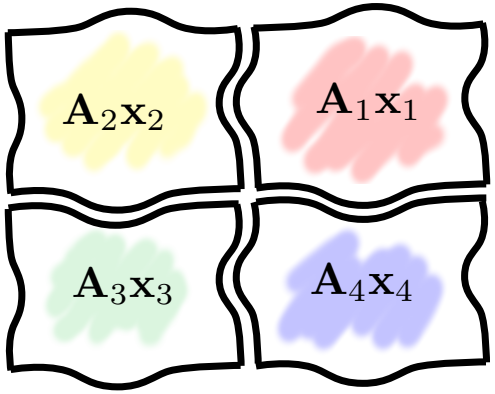
# Parallel *vs* parallelized solvers

**Parallelized solvers**: Krylov methods (GMRES, etc.)

Scalar-product and matrix vectors are parallel

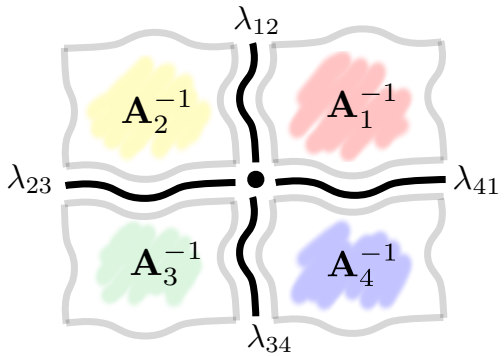
$$\mathbf{x} \cdot \mathbf{y} = \mathop{\mathbf{A}}_{i} \left( \mathbf{x}'_i \cdot \mathbf{y}'_i \right)$$

$$\mathbf{A}\mathbf{x} = \mathop{\mathbf{A}}_{i} \mathbf{A}_i \mathbf{x}_i$$

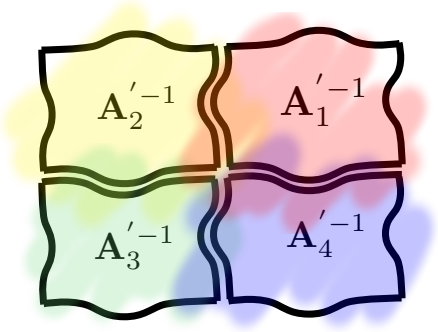



**Parallel solvers**: Schur complement, Schwarz

*Non-overlapping*: Schur complement, block LU



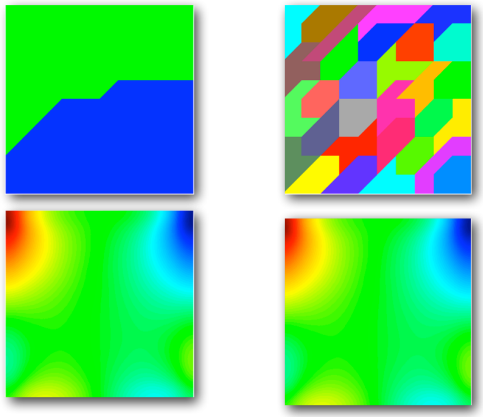

*Overlapping*: Schwarz

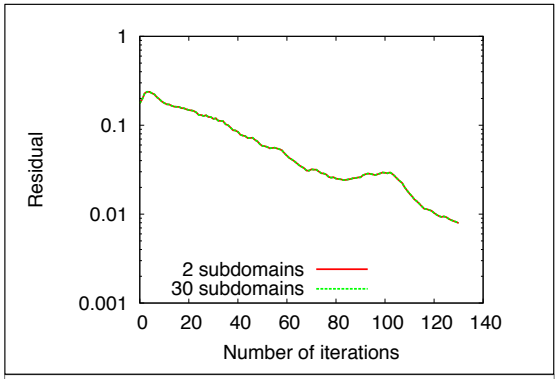


# Parallel *vs* parallelized solvers

## Parallelized solvers

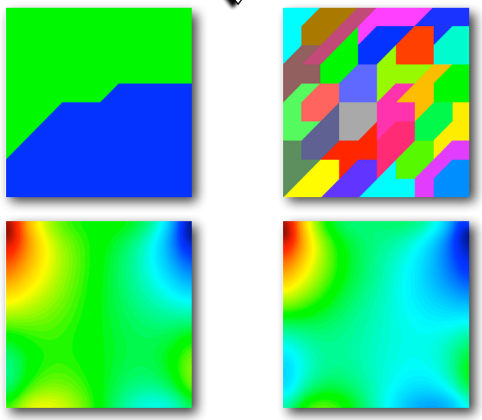
Same # iterations in sequential and parallel

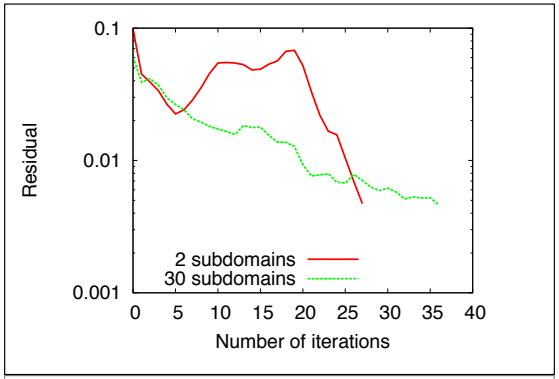





## Parallel solvers

Different # iterations for different # subdomains





# Rumor

- Domain decomposition for mathematicians
- Krylov solvers for engineers

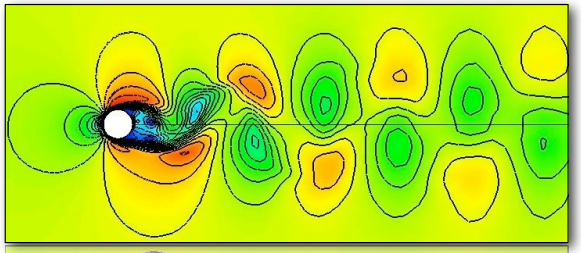
Why?

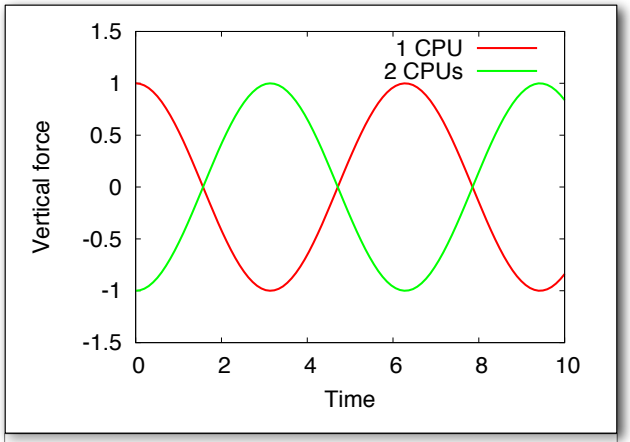
Engineers prefer the convergence to be the same for any # CPUs

Why?

# A fact's a fact

Why engineers don't like that convergence depends on # of subdomains






Reproducibility may be important

# Parallelized solvers with parallel preconditioner

*A good compromise*

Parallelized solver

$\mathbf{M}^{-1}\mathbf{Ax} = \mathbf{M}^{-1}\mathbf{b}$: GMRES, CG

+

Parallel preconditioner (local solver)

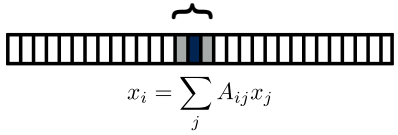
$\mathbf{M}$: Schwarz, block LU

+

Coarse grid correction

# Parallelized solvers with parallel preconditioner

zone of influence of black node
through matrix-vector product



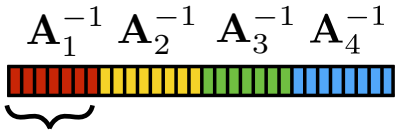

Simple iteration

$$\mathbf{x}^{k+1} = \mathbf{x}^k + (\mathbf{b} - \mathbf{A}\mathbf{x}^k)$$

Good scalability

High frequencies

Local solver




zone of influence of a red node
on others after preconditioning

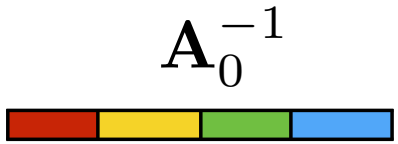
$$\mathbf{x}^{k+1} = \mathbf{x}^k + \begin{pmatrix} \mathbf{A}_1^{-1} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \mathbf{A}_N^{-1} \end{pmatrix} (\mathbf{b} - \mathbf{A}\mathbf{x}^k)$$

Perfect scalability

Better conditioning

CPU consuming

Coarse grid correction




$$\mathbf{x}^{k+1} = \mathbf{x}^k + \left[ \mathbf{R}_0^t \mathbf{A}_0^{-1} R_0 + \begin{pmatrix} \mathbf{A}_1^{-1} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \mathbf{A}_N^{-1} \end{pmatrix} \right] (\mathbf{b} - \mathbf{A}\mathbf{x}^k)$$

$\mathbf{A}_0^{-1}$ hard to parallelize

Couple subdomains

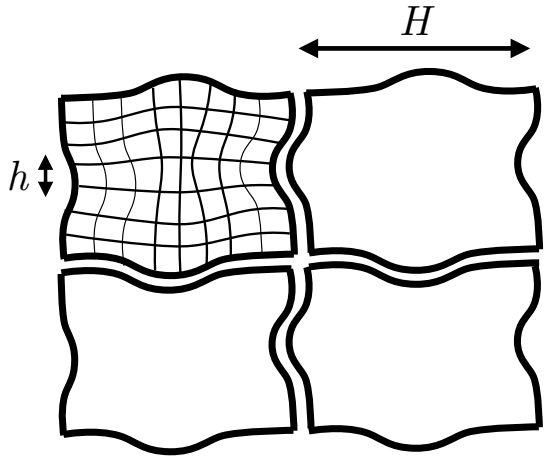
# What is the ideal solver?

**Requirements**

CPU time not affected when mesh size ($h$) decreases
CPU time not affected when subdomain size ($H$) decreases

$\Leftrightarrow$

CPU time not affected when # of degrees of freedom ($dof=1/h$) increases
CPU time not affected when # of CPUs ($np=1/H$) increases

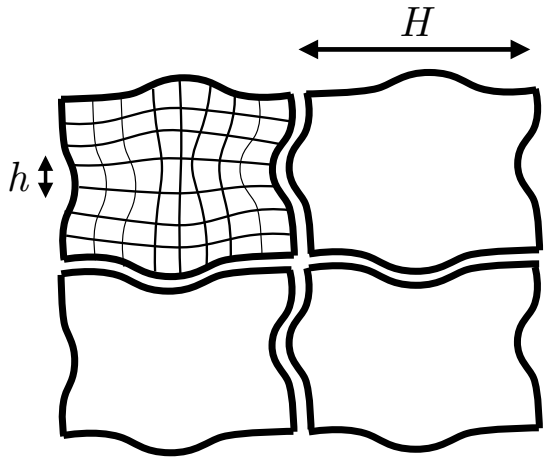


# Strong vs weak scalability

**Definitions**

**Strong scalability**= how the solution time varies with the number of processors for a fixed total problem size ($dof$)
(keeping $h$ constant)

**Weak scalability**= how the solution time varies with the number of processors for a fixed problem size per CPU
(keeping $H/h = dof/np$ constant)

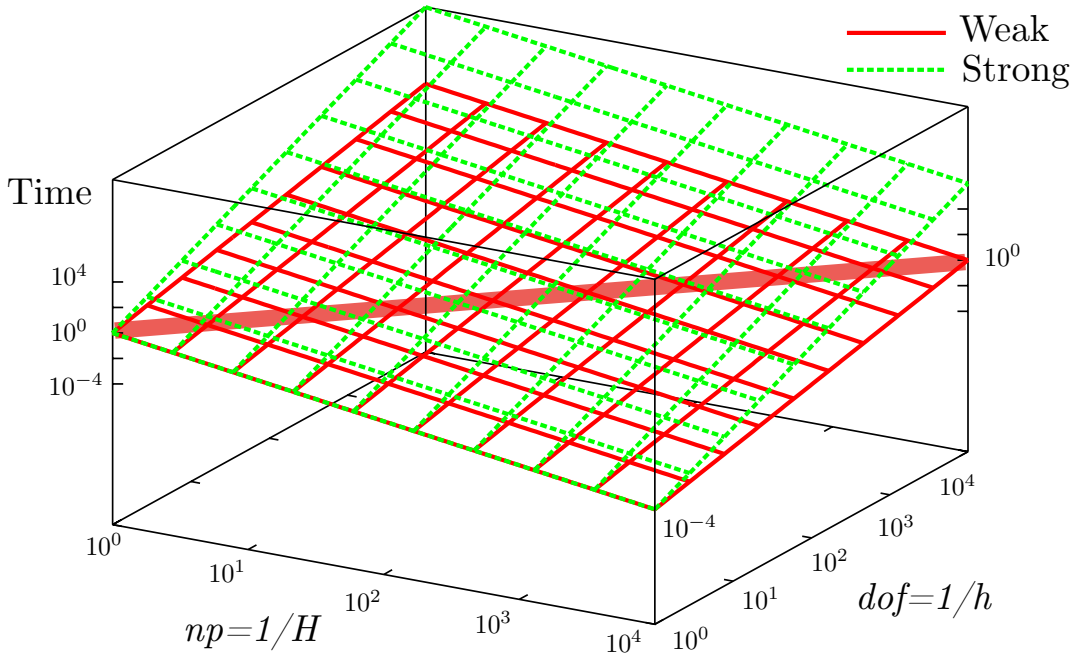


# Strong vs weak scalability

**Efficiency**

Strong scalability$= \dfrac{t_1(dof)}{np \times t_{np}(dof)}$ : How much faster can I go to solve the same problem on more CPUs?

Weak scalability$= \dfrac{t_1(dof)}{t_{np}(np \times dof)}$ : Can I refine the mesh without altering the CPU time by adding more CPUs?



# Dreams *vs* reality

- Weak scalability hardly exist
- At most, weak scalability is satisfied using number of iterations instead of CPU time!

Olivier Dubois[1] and Martin J. Gander[2]

| $\mathbf{M_x \times M_y}$ | $2\times2$ | $4\times4$ | $6\times6$ | $8\times8$ | $9\times9$ |
|---|---|---|---|---|---|
| no. of unknowns | 147,456 | 589,824 | 1,327,104 | 2,359,296 | 2,985,984 |
| | Stationary iterative method | | | | |
| RAS2 | 439 | 1082 | 1528 | 1557 | 1798 |
| ORAS2 | 325 | 316 | 323 | 332 | 324 |
| | Preconditioned GMRES | | | | |
| RAS2 | 40 | 47 | 48 | 48 | 48 |
| ORAS2 | 18 | 20 | 21 | 21 | 21 |

**Table 2.** Number of iterations for a weak scaling experiment.

Junxian Wang[1] Shi Shu[1] and Liuqiang Zhong[2]

**Table 1.** No. of iterations for Example 1 with TLB-AMG-CG-p.

| $n(m)$ | 3(4) | | | 3(6) | | |
|---|---|---|---|---|---|---|
| $\beta_0$ | $10^{-5}$ | 1 | $10^5$ | $10^{-5}$ | 1 | $10^5$ |
| $k=2$ | 4 | 4 | 5 | 4 | 5 | 6 |
| $k=3$ | 5 | 5 | 6 | 5 | 5 | 6 |
| $k=4$ | 5 | 6 | 9 | 5 | 5 | 9 |

! With this criterion, a LU would be a perfect weak scalable solver which always converges in 1 iteration!

- Why weak scalability so difficult to obtain?

1. Condition number of matrix increases with increasing *dof* $\Longrightarrow$ Exact local solver
2. Parallel solver performance decreases with increasing *np* $\Longrightarrow$ Coarse grid solver

# Principle

# of

# *Conservation of Problems*

We can lower the # iterations

but

at the expense of CPU time

# But...

Sometimes, smart tricks can help

# If...

if one stays close to the *physics*

# Special case: Navier-Stokes

Monolithic solution of Navier-Stokes

$$\mathbf{M}^{-1} \begin{bmatrix} \mathbf{A}_{uu} & \mathbf{A}_{up} \\ \mathbf{A}_{pu} & \mathbf{A}_{pp} \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ \mathbf{p} \end{bmatrix} = \mathbf{M}^{-1} \begin{bmatrix} \mathbf{b}_u \\ \mathbf{b}_p \end{bmatrix}$$

Difficult to converge unless one considers a complex preconditioner $\mathbf{M}$

A complex preconditioner does not parallelize well

Its cost is high

## Special case: Navier-Stokes

For years

fractional step methods

have proved to be a nice alternative to monolithic schemes

to solve incompressible flows

# Special case: Navier-Stokes

## The strong *vs* the algebraic vision

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\nabla p + \frac{1}{\mathrm{Re}} \nabla^2 \mathbf{u},$$

$$\nabla \cdot \mathbf{u} = 0.$$

$$\frac{1}{\Delta t}\left[I - \frac{\Delta t}{2\,\mathrm{Re}} \nabla^2\right] \mathbf{u}^* = \frac{1}{\Delta t}\left[I + \frac{\Delta t}{2\,\mathrm{Re}} \nabla^2\right] \mathbf{u}^n - \left[\frac{3}{2}(\mathbf{u}^n \cdot \nabla)\,\mathbf{u}^n - \frac{1}{2}(\mathbf{u}^{n-1} \cdot \nabla)\,\mathbf{u}^{n-1}\right]$$

$$\Delta t (\nabla \cdot \nabla)\, p^{n+1} = \nabla \cdot \mathbf{u}^*,$$

$$\mathbf{u}^{n+1} = \mathbf{u}^* - \Delta t \nabla p^{n+1}$$

# Special case: Navier-Stokes

Stabilized FEM

$$\begin{bmatrix} \mathbf{A}_{uu} & \mathbf{A}_{up} \\ \mathbf{A}_{pu} & \mathbf{A}_{pp} \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ \mathbf{p} \end{bmatrix} = \begin{bmatrix} \mathbf{b}_u \\ \mathbf{b}_p \end{bmatrix}$$

*Monolithic*

$$\mathbf{M}^{-1} \begin{bmatrix} \mathbf{A}_{uu} & \mathbf{A}_{up} \\ \mathbf{A}_{pu} & \mathbf{A}_{pp} \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ \mathbf{p} \end{bmatrix} = \mathbf{M}^{-1} \begin{bmatrix} \mathbf{b}_u \\ \mathbf{b}_p \end{bmatrix}$$

*Fractional scheme*

Split momentum and continuity

# Special case: Navier-Stokes

$$\rho\frac{\partial \boldsymbol{u}}{\partial t} + \rho(\boldsymbol{u}\cdot\nabla)\boldsymbol{u} - \nabla\cdot[2\mu\boldsymbol{\varepsilon}(\boldsymbol{u})] + \nabla p = \rho\boldsymbol{f}$$
$$\nabla\cdot\boldsymbol{u} = 0$$

Discretize

$$\begin{bmatrix} \mathbf{A}_{uu} & \mathbf{A}_{up} \\ \mathbf{A}_{pu} & \mathbf{A}_{pp} \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ \mathbf{p} \end{bmatrix} = \begin{bmatrix} \mathbf{b}_u \\ \mathbf{b}_p \end{bmatrix}$$

Eliminate velocity

Pressure Schur complement system

$$\mathbf{S}\mathbf{p} = \mathbf{b}_s$$
$$\mathbf{S} := \mathbf{A}_{pp} - \mathbf{A}_{pu}\mathbf{A}_{uu}^{-1}\mathbf{A}_{up}$$
$$\mathbf{b}_s := \mathbf{b}_p - \mathbf{A}_{pu}\mathbf{A}_{uu}^{-1}\mathbf{b}_u$$

Define

$$\mathbf{r}_k = \mathbf{b}_s - \mathbf{S}\mathbf{p}^k$$
$$\mathbf{Q} \approx \mathbf{S}$$
$$\alpha = \frac{< \mathbf{r}^k, \mathbf{S}\mathbf{Q}^{-1}\mathbf{r}^k >}{< \mathbf{S}\mathbf{Q}^{-1}\mathbf{r}^k, \mathbf{S}\mathbf{Q}^{-1}\mathbf{r}^k >}$$

Simple iteration

Orthomin(1)

$$\mathbf{p}^{k+1} = \mathbf{p}^k + \alpha\mathbf{Q}^{-1}\mathbf{r}^k$$

Solve

2 momentum solves
1 continuity solve

$$\mathbf{A}_{uu}\mathbf{u}^{k+1} = \mathbf{b}_u - \mathbf{A}_{up}\mathbf{p}^k$$
$$\mathbf{Q}\mathbf{z} = \mathbf{r}^k$$
$$\mathbf{A}_{uu}\mathbf{v} = \mathbf{A}_{up}\mathbf{z}$$

Update

$$\begin{cases} \mathbf{p}^{k+1} = \mathbf{p}^k + \alpha\mathbf{z} \\ \mathbf{u}^{k+2} = \mathbf{u}^{k+1} - \alpha\mathbf{v} \end{cases}$$

Next linearization step

$$\ldots$$

# Special case: Navier-Stokes

*Monolithic*

$$\mathbf{M}^{-1}\begin{bmatrix}\mathbf{A}_{uu} & \mathbf{A}_{up}\\ \mathbf{A}_{pu} & \mathbf{A}_{pp}\end{bmatrix}\begin{bmatrix}\mathbf{u}\\ \mathbf{p}\end{bmatrix} = \mathbf{M}^{-1}\begin{bmatrix}\mathbf{b}_u\\ \mathbf{b}_p\end{bmatrix}$$

Complex preconditioner

*vs*

Simple preconditioner

*vs*

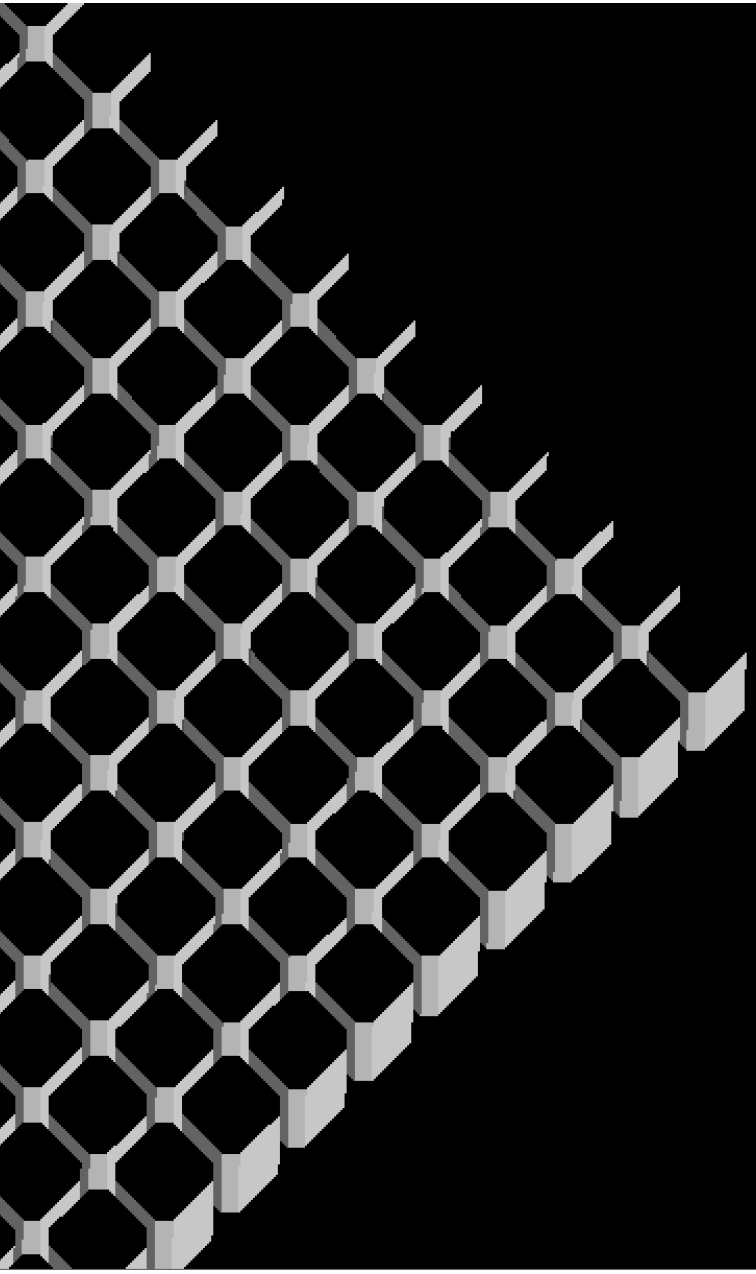
One system

*vs*

Three coupled systems

*Fractional scheme*

$$\begin{aligned}\mathbf{M}_u^{-1}\mathbf{A}_{uu}\mathbf{u}^{k+1} &= \mathbf{M}_u^{-1}(\mathbf{b}_u - \mathbf{A}_{up}\mathbf{p}^k)\\ \mathbf{M}_p^{-1}\mathbf{Q}\mathbf{z} &= \mathbf{M}_p^{-1}\mathbf{r}^k\\ \mathbf{M}_u^{-1}\mathbf{A}_{uu}\mathbf{v} &= \mathbf{M}_u^{-1}\mathbf{A}_{up}\mathbf{z}\end{aligned}$$



# Basic operations

$\mathbf{Ax}$

$\mathbf{x} \cdot \mathbf{y}$

# Basic operations of a solver

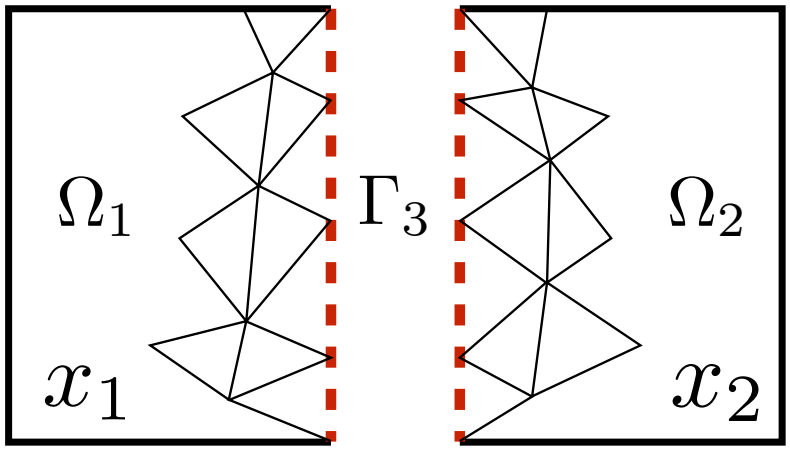
Matrix-vector

$$
\begin{aligned}
&\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0 \\
&\mathbf{z}_0 = \mathbf{M}^{-1}\mathbf{r}_0 \\
&\mathbf{p}_0 = \mathbf{z}_0 \\
&\text{For } j = 0, 1, \text{ until convergence Do:} \\
&\quad \mathbf{v}_j = \mathbf{A}\mathbf{p}_j \\
&\quad \alpha_j = (\mathbf{r}_j, \mathbf{z}_j)/(\mathbf{v}, \mathbf{p}_j) \\
&\quad \mathbf{x}_{j+1} = \mathbf{x}_j + \alpha_j \mathbf{p}_j \\
&\quad \mathbf{r}_{j+1} = \mathbf{r}_j - \alpha_j \mathbf{v}_j \\
&\quad \mathbf{z}_{j+1} = \mathbf{M}^{-1}\mathbf{r}_{j+1} \\
&\quad \beta_j = (\mathbf{r}_{j+1}, \mathbf{z}_{j+1})/(\mathbf{r}_j, \mathbf{z}_j) \\
&\quad \mathbf{p}_{j+1} = \mathbf{z}_{j+1} + \beta_j \mathbf{p}_j \\
&\text{EndDo}
\end{aligned}
$$

Scalar product

# Matrix-vector product: notations

*Finite element context*

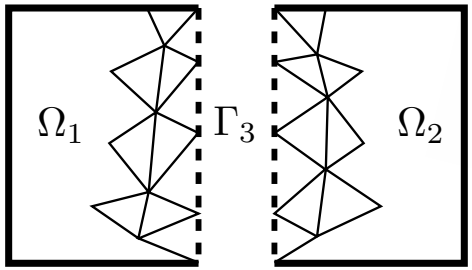



$x_3^{(i)}$ = local interface value in subdomain $i$
Can have $x_3^{(1)} \neq x_3^{(2)}$

$x_3$ = global interface value
Same in both subdomains $x_3^{(1)} = x_3^{(2)} = x_3$

# Matrix-vector product

- Global matrix has a block structure $\mathbf{A} = \begin{pmatrix} A_{11} & 0 & A_{13} \\ 0 & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix}$




- It can be computed independently as

$$A_1 = \begin{pmatrix} A_{11} & A_{13} \\ A_{31} & A_{33}^{(1)} \end{pmatrix} \qquad A_2 = \begin{pmatrix} A_{22} & A_{23} \\ A_{32} & A_{33}^{(2)} \end{pmatrix}$$

Blocks $A_{33}^{(1)}$ and $A_{33}^{(2)}$ represent the interactions of the interface dof and: $A_{33} = A_{33}^{(1)} + A_{33}^{(2)}$

- Using block notation, matrix-vector $\mathbf{y} = \mathbf{A}\mathbf{x}$ product reads

$$\begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} A_{11} & 0 & A_{13} \\ 0 & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} A_{11}x_1 + A_{13}x_3 \\ A_{22}x_2 + A_{23}x_3 \\ A_{31}x_1 + A_{32}x_2 + A_{33}x_3 \end{pmatrix}$$

- Which can be computed with two independent matrix-vector products:

$$\begin{pmatrix} y_1 \\ y_3^{(1)} \end{pmatrix} = \begin{pmatrix} A_{11}x_1 + A_{13}x_3 \\ A_{31}x_1 + A_{33}^{(1)}x_3 \end{pmatrix}$$

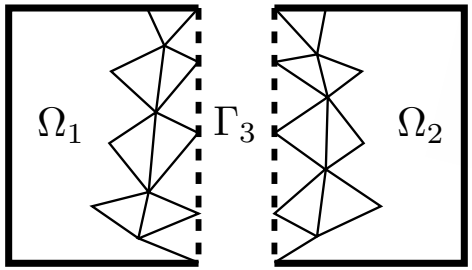
$$\begin{pmatrix} y_2 \\ y_3^{(2)} \end{pmatrix} = \begin{pmatrix} A_{22}x_2 + A_{23}x_3 \\ A_{32}x_2 + A_{33}^{(2)}x_3 \end{pmatrix}$$

with

$$\begin{aligned} A_{33} &= A_{33}^{(1)} + A_{33}^{(2)} \\ y_3 &= y_3^{(1)} + y_3^{(2)} \end{aligned}$$

# Matrix-vector product

*Parallel implementation*




- Assemble local matrices $A_1 = \begin{pmatrix} A_{11} & A_{13} \\ A_{31} & A_{33}^{(1)} \end{pmatrix}$ $A_2 = \begin{pmatrix} A_{22} & A_{23} \\ A_{32} & A_{33}^{(2)} \end{pmatrix}$
- Compute local matrix-vector products

$$\begin{pmatrix} y_1 \\ y_3^{(1)} \end{pmatrix} = \begin{pmatrix} A_{11}x_1 + A_{13}x_3 \\ A_{31}x_1 + A_{33}^{(1)}x_3 \end{pmatrix}$$

$$\begin{pmatrix} y_2 \\ y_3^{(2)} \end{pmatrix} = \begin{pmatrix} A_{22}x_2 + A_{23}x_3 \\ A_{32}x_2 + A_{33}^{(2)}x_3 \end{pmatrix}$$

- Exchange local result

$$\Omega_1 \xrightarrow{y_3^{(1)}} \Omega_2$$
$$\Omega_2 \xrightarrow{y_3^{(2)}} \Omega_1$$

`MPI_SendRecv`

- Assemble result

$$\Omega_1 : \quad y_3 = y_3^{(1)} + y_3^{(2)}$$
$$\Omega_2 : \quad y_3 = y_3^{(2)} + y_3^{(1)}$$

# Matrix-vector product

! IMPORTANT

- The interface values $x_3$ must be "global", i.e. the same on both sides of the interface $x_3^{(1)} = x_3^{(2)} = x_3$

$$\begin{pmatrix} y_1 \\ y_3^{(1)} \end{pmatrix} = \begin{pmatrix} A_{11}x_1 + A_{13}x_3 \\ A_{31}x_1 + A_{33}^{(1)}x_3 \end{pmatrix}$$

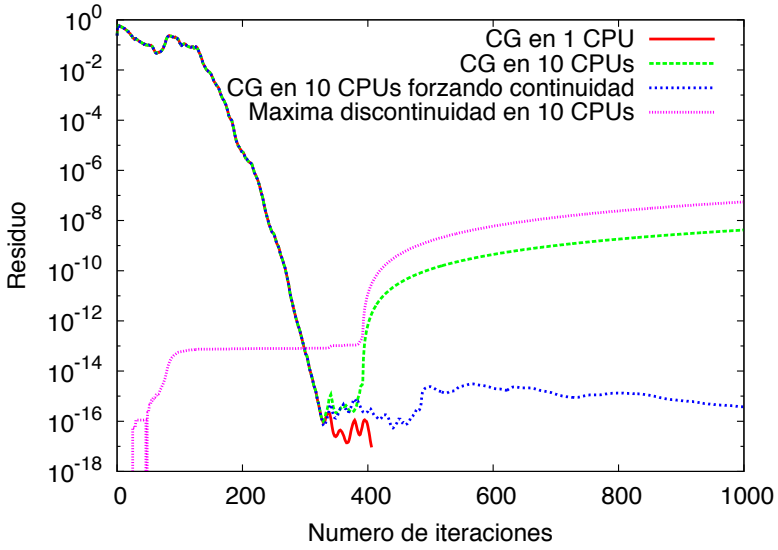
$$\begin{pmatrix} y_2 \\ y_3^{(2)} \end{pmatrix} = \begin{pmatrix} A_{22}x_2 + A_{23}x_3 \\ A_{32}x_2 + A_{33}^{(2)}x_3 \end{pmatrix}$$

- But continuity of $y_3$ is not enforced explicitly!

- Ideally, operations should be carried out in the same way

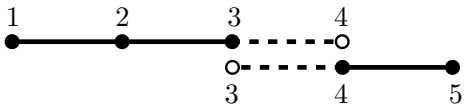
$$\Omega_1 : \quad y_3 = y_3^{(1)} + y_3^{(2)}$$
$$\Omega_2 : \quad y_3 = \cancel{y_3^{(2)}} + y_3^{(1)} \quad \Leftarrow \quad y_3 = y_3^{(1)} + y_3^{(2)}$$

to avoid round-off errors



# Matrix-vector product

## Finite Difference




1 Exchange: $\Omega_1$ sends $x_3$ to $\Omega_2$

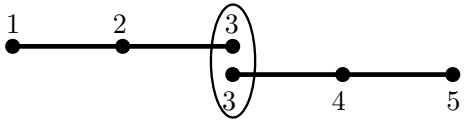
2 Exchange: $\Omega_2$ sends $x_4$ to $\Omega_1$

3 Local matrix-vector product

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & & \\ 1 & 1 & 1 & \\ & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

$$\begin{bmatrix} y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} 2 & 2 & 2 \\ & 2 & 2 \end{bmatrix} \begin{bmatrix} x_3 \\ x_4 \\ x_5 \end{bmatrix}$$

## Finite Element




1 Local matrix-vector product

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3^{(1)} \end{bmatrix} = \begin{bmatrix} 1 & 1 & \\ 1 & 1 & 1 \\ & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

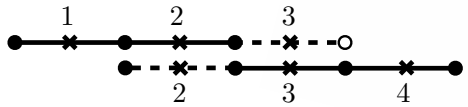
$$\begin{bmatrix} y_3^{(2)} \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} 2 & 2 & \\ 2 & 2 & 2 \\ & 2 & 2 \end{bmatrix} \begin{bmatrix} x_3 \\ x_4 \\ x_5 \end{bmatrix}$$

2 Exchange: $\Omega_1$ sends $y_3^{(1)}$ to $\Omega_2$

3 Exchange: $\Omega_2$ sends $y_3^{(2)}$ to $\Omega_1$

4 Assembly: $y_3 = y_3^{(1)} + y_3^{(2)}$

## Finite Volume




1 Exchange: $\Omega_1$ sends $x_2$ to $\Omega_2$

2 Exchange: $\Omega_2$ sends $x_3$ to $\Omega_1$

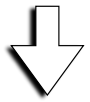
3 Local matrix-vector product

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 & \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

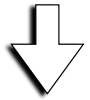
$$\begin{bmatrix} y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} 2 & 2 & 2 \\ & 2 & 2 \end{bmatrix} \begin{bmatrix} x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

# Scalar product

$$\alpha = \mathbf{x} \cdot \mathbf{y}$$



Each subdomain $i$ computes a local scalar product $\alpha_i$



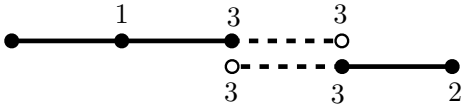
`MPI_AllReduce`

$$\alpha = \sum_{i=1}^{N} \alpha_i$$

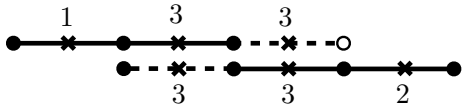
What should be $\alpha_i$?

# Scalar product

## Finite difference




## Finite volume




FD, FV: only one subdomain is responsible for interface each node (own)

⇨ Scalar product is computed naturally as follows:

$$\alpha_1 = (x_1 \cdot y_1) + (x^{(1)}_{3,\text{own}} \cdot y^{(1)}_{3,\text{own}})$$
$$\alpha_2 = (x_2 \cdot y_2) + (x^{(2)}_{3,\text{own}} \cdot y^{(2)}_{3,\text{own}})$$
$$\alpha_1 + \alpha_2 = (x_1 \cdot y_1) + (x^{(1)}_{3,\text{own}} \cdot y^{(1)}_{3,\text{own}}) + (x_2 \cdot y_2) + (x^{(2)}_{3,\text{own}} \cdot y^{(2)}_{3,\text{own}}) = \alpha$$

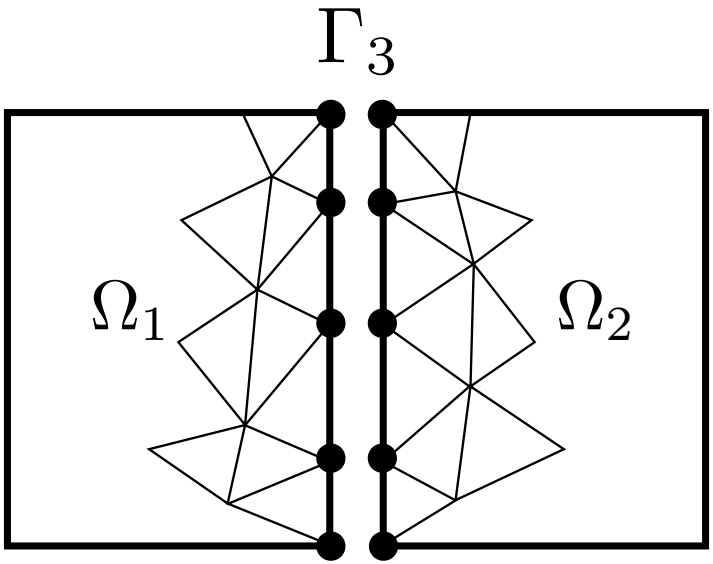
# Scalar product

## Finite element

$$\Omega_1 : \quad \alpha_1 = \begin{pmatrix} x_1 \\ x_3 \end{pmatrix} \cdot \begin{pmatrix} y_1 \\ y_3 \end{pmatrix} = (x_1 \cdot y_1) + (x_3 \cdot y_3)$$

$$\Omega_2 : \quad \alpha_2 = \begin{pmatrix} x_2 \\ x_3 \end{pmatrix} \cdot \begin{pmatrix} y_2 \\ y_3 \end{pmatrix} = (x_2 \cdot y_2) + (x_3 \cdot y_3)$$

$$\begin{aligned} \alpha &= \alpha_1 + \alpha_2 \\ &= (x_1 \cdot y_1) + (x_2 \cdot y_2) + \boxed{2\ (x_3 \cdot y_3)} \end{aligned}$$

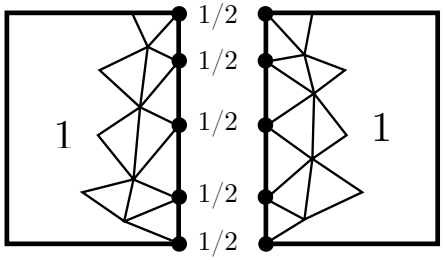



! Interface nodes are duplicated

# Scalar product

1. *By weight*

$$\alpha_1 = 1\,(x_1 \cdot y_1) + \frac{1}{2}\,(x_3 \cdot y_3)$$
$$\alpha_2 = 1\,(x_2 \cdot y_2) + \frac{1}{2}\,(x_3 \cdot y_3)$$




2. *By distributivity*

$$\alpha_1 = (x_1 \cdot y_1) + (x_3 \cdot y_3^{(1)})$$
$$\alpha_2 = (x_2 \cdot y_2) + (x_3 \cdot y_3^{(2)})$$

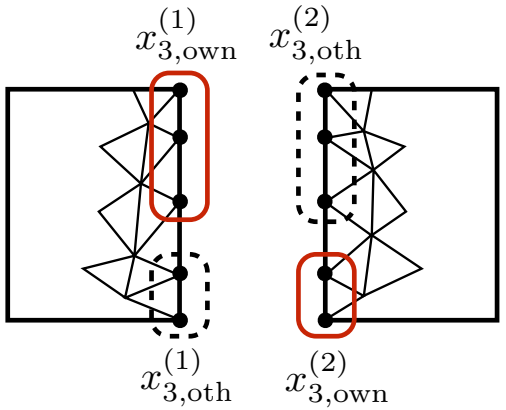
useful when

$$\begin{pmatrix} y_1 \\ y_3^{(1)} \end{pmatrix} = \begin{pmatrix} A_{11}x_1 + A_{13}x_3 \\ A_{31}x_1 + A_{33}^{(1)}x_3 \end{pmatrix}$$
$$\begin{pmatrix} y_2 \\ y_3^{(2)} \end{pmatrix} = \begin{pmatrix} A_{22}x_2 + A_{23}x_3 \\ A_{32}x_2 + A_{33}^{(2)}x_3 \end{pmatrix}$$

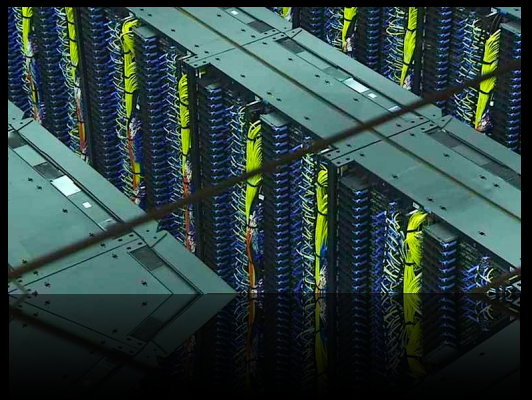
3. *By property*

⇨ Divide interface in disjoint parts: own (own) and other (oth)

$$\alpha_1 = (x_1 \cdot y_1) + (x_{3,\text{own}}^{(1)} \cdot y_{3,\text{own}}^{(1)})$$
$$\alpha_2 = (x_2 \cdot y_2) + (x_{3,\text{own}}^{(2)} \cdot y_{3,\text{own}}^{(2)})$$
$$\alpha_1 + \alpha_2 = (x_1 \cdot y_1) + (x_{3,\text{own}}^{(1)} \cdot y_{3,\text{own}}^{(1)}) + (x_2 \cdot y_2) + (x_{3,\text{own}}^{(2)} \cdot y_{3,\text{own}}^{(2)}) = \alpha$$



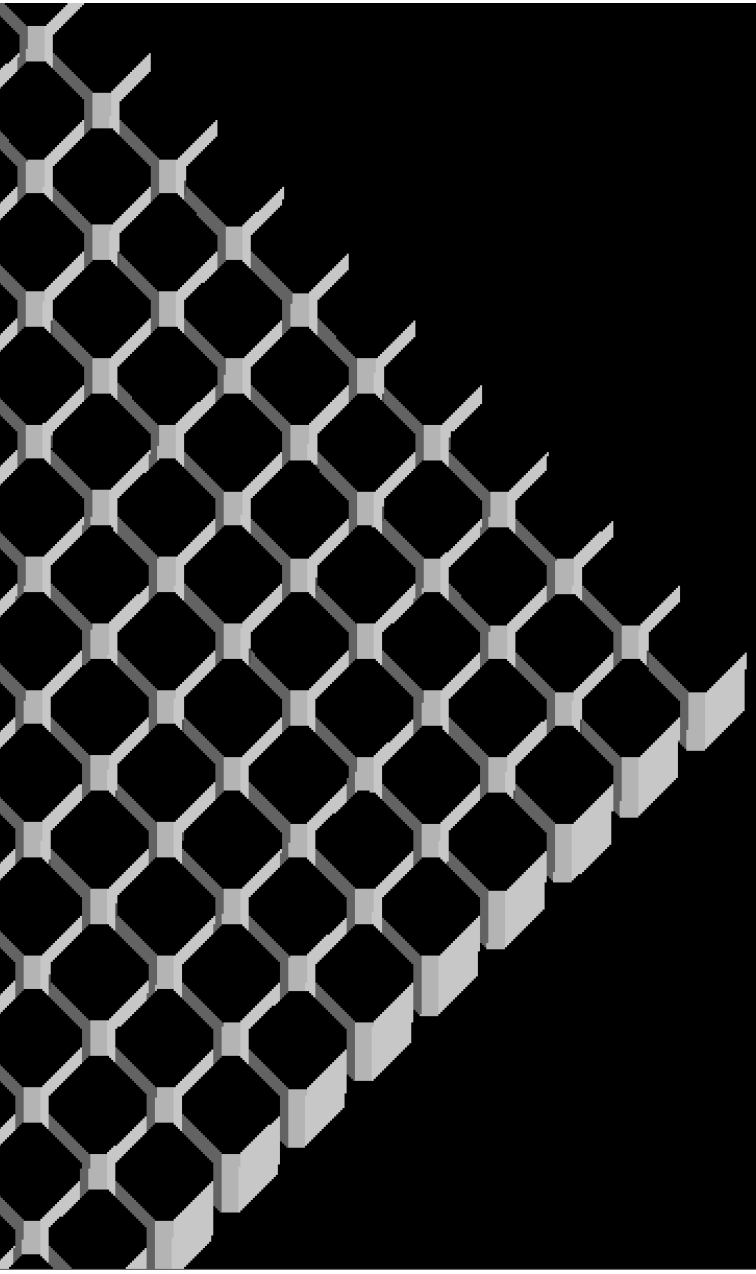
# II. Implementation



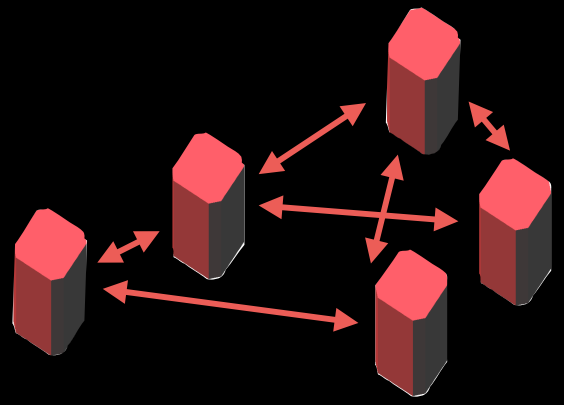
# From experience...

To implement an algorithm helps its understanding

This fact is stronger for parallel algorithm



# MPI point-to-point communications



# Point-to-point communication

Subdomains have to exchange arrays with their neighbors

1. Without scheduling ⇨ `MPI_ISend+MPI_IRecv`

   Loop over the neighbors in a **non-ordered** way

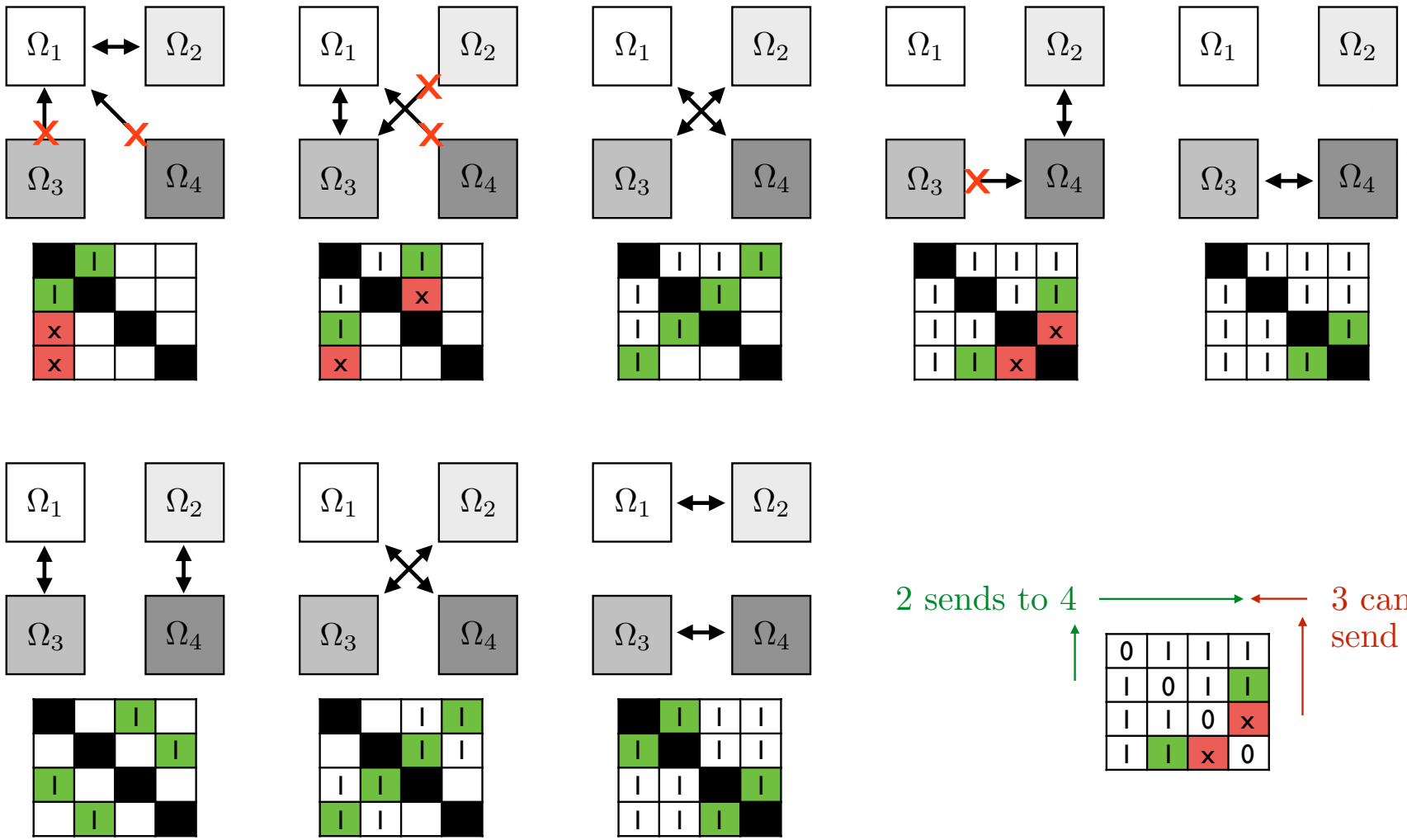
2. With scheduling

   Loop over the neighbors in an **ordered** way

   1. Blocking ⇨ `MPI_SendRecv`

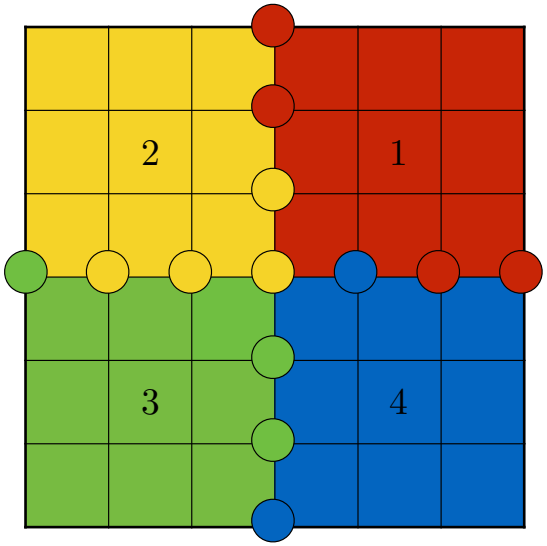
   2. Non-blocking ⇨ `MPI_ISend+MPI_IRecv`

# Scheduling

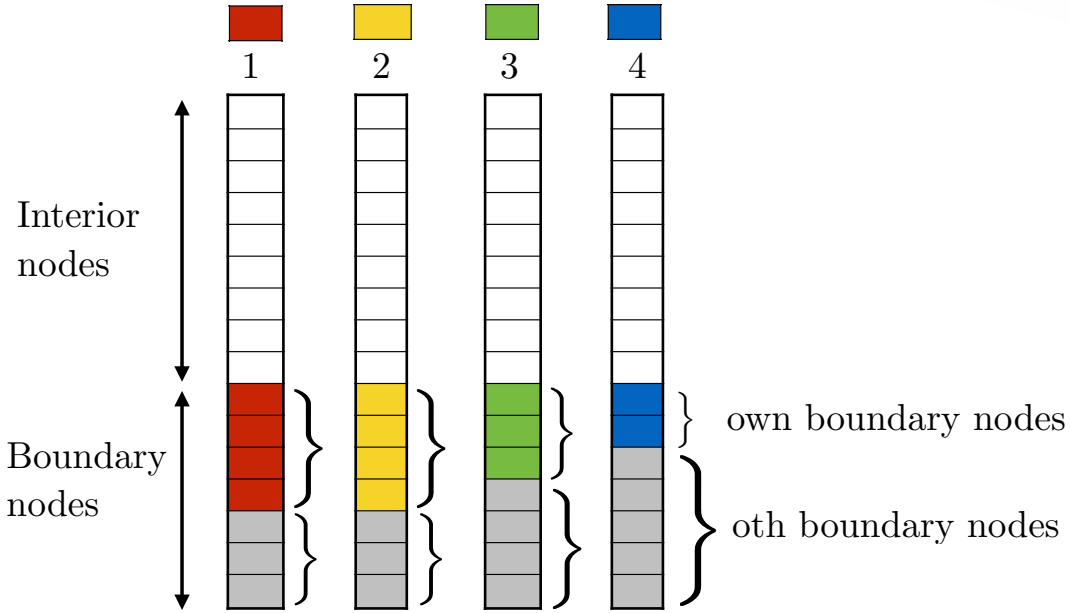


# Node ordering

*Mesh*




*Node ordering*



# Algorithm: point-to-point communication

## Exchange array $y$ (e.g. $y$=result of a matrix-vector)

*1. Local matrix-vector* $y = A\ x$

*2. Local copy*

do $i$=1,$nb\_neighbors$
  $temp\_send_i(1{:}n_i) = y(list_i(1{:}n_i))$
end do

*3. MPI_SendRecv*

do $j$=1,$nb\_neighbors$
  $i$=$domain(j)$
  `MPI_SendRecv` $tem\_send_i$
    and $tem\_recv_i$ with $i$
end do

- Scheduling: subd. $i$ is $j^{th}$ to communicate with
- No Scheduling: $domain(i)$=$i$

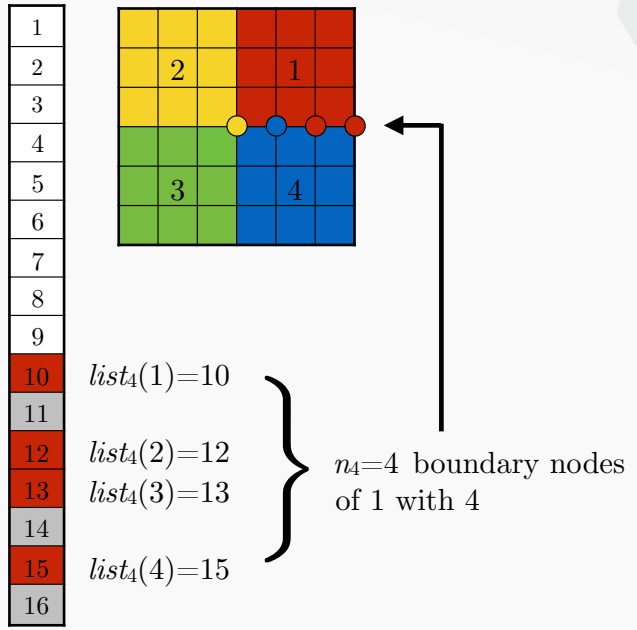
*4. Assemble*

do $i$=1,$nb\_neighbors$
  $y(list_i(1{:}n_i)) = y(list_i(1{:}n_i)) + temp\_recv_i(1{:}n_i)$
end do

$$y_3 = y_3^{(1)} + y_3^{(2)}$$

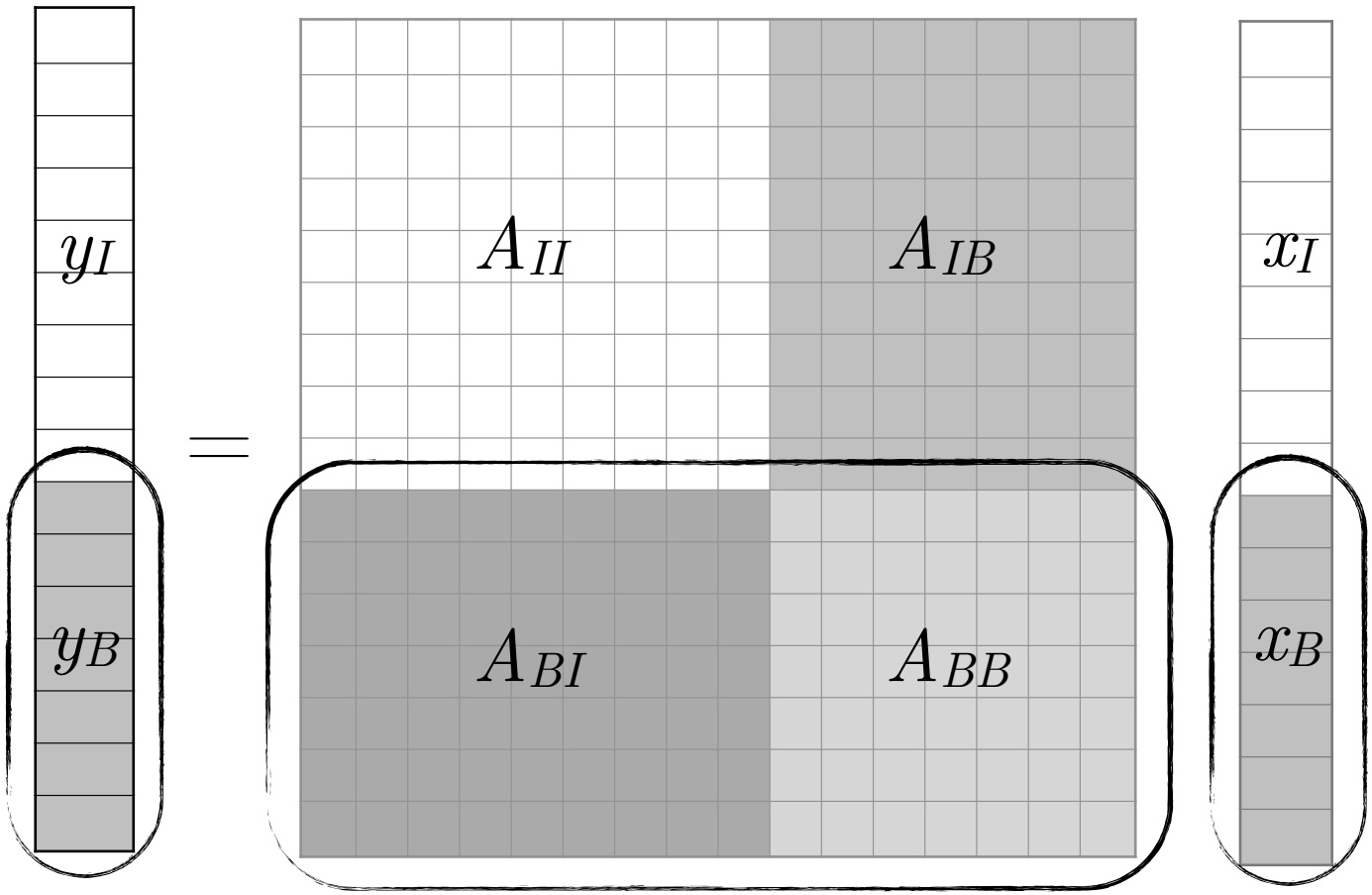
### Example subdomain 1

Communicate with 2,4,3
$nb\_neighbors$=3
$domain(1)$=2
$domain(2)$=4
$domain(3)$=3

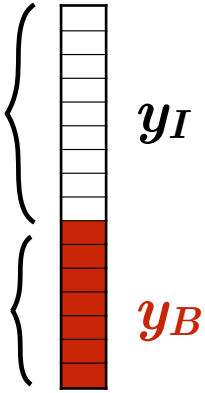



- $nb\_neighbors$ = nb of neighbors
- $n_i$ = nb of boundary nodes with neighbor $i$
- $list_i(k)$ = local numbering of boundary nodes with subdomain $i$
- $domain(i)$ = $i^{th}$ neighbor

# Why matrix-vector can be done asynchronously



# Point-to-point communication synch *vs* asynch




overlap work & communication

## Asynchronous

***1. Local boundary matrix-vector $y_B = A\ x$***

*2. Local copy*
do $i$=1,$nb\_neighbors$
  $temp\_send_i(1{:}n_i) = y(list_i(1{:}n_i))$
end do

*3. MPI_ISend / IRecv*
do $j$=1,$nb\_neighbors$
  $i$=$domain(j)$
  `MPI_ISend` $tem\_send_i$ to i
  `MPI_IRecv` $tem\_recv_i$ from $i$
end do

***4. Local interior matrix-vector $y_I = A\ x$***

***5. MPI_ Waitall***

*6. Assembly*
do $i$=1,$nb\_neighbors$
  $y(list_i(1{:}n_i)) = y(list_i(1{:}n_i)) + temp\_recv_i(1{:}n_i)$
end do

Communication

## Synchronous

*1. Local matrix-vector $y = A\ x$*

*2. Local copy*
do $i$=1,$nb\_neighbors$
  $temp\_send_i(1{:}n_i) = y(list_i(1{:}n_i))$
end do

*3. MPI_SendRecv*
do $j$=1,$nb\_neighbors$
  $i$=$domain(j)$
  `MPI_SendRecv` $tem\_send_i$ and $tem\_recv_i$ with $i$
end do

*4. Assembly*
do $i$=1,$nb\_neighbors$
  $y(list_i(1{:}n_i)) = y(list_i(1{:}n_i)) + temp\_recv_i(1{:}n_i)$
end do


Alya

# Where are we?

*HPC Simulation Tools:* Alya

**Alya is one of the two CFD codes of the PRACE benchmark suite**

respective user communities, as well the coverage of scientific a
a final list of 12 codes to form the initial version of UEABS, wh

| | |
|---|---|
| **Particle Physics**: | QCD |
| **Classical MD**: | NAMD, GROMACS |
| **Quantum MD**: | Quantum Espresso, CP2K, GPAW |
| **CFD**: | Code_Saturne, ALYA |
| **Earth Sciences**: | NEMO, SPECFEM3D |
| **Plasma Physics**: | GENE |
| **Astrophysics**: | GADGET |

Available online at www.prace-ri.eu

**Partnership for Advanced Computing in Europe**

Selection of a Unified European Application Benchmark Suite

J. Mark Bull[a]*, Andrew Emerson[b]

[a]EPCC, University of Edinburgh, King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, UK.
[b]CINECA, via Magnanelli 6/3, 40033 Casalecchio di Reno, Bologna, Italy.

Lindgren (S), Cray XE system at PDC, incompressible flow 12288 CPU's (collaboration with Jing Gong from PDC)

Huygens, (NL), IBM power 6, incompressible flow, 2128 CPU's

Jugene BG (D): 16384 CPU's, incompressible flow (PRACE project for Mesh multiplication) and, running first tests of FSI in collaboration with Paolo Crosetto (Jülich)

Fermi BG (I): 16384 CPU's, incompressible flow + species transport + Lagrangian particles (PRACE project for nose)

Curie Bullx (F): 22528 CPU's, incompressible flow (collaboration with Jing Gong - PDC)

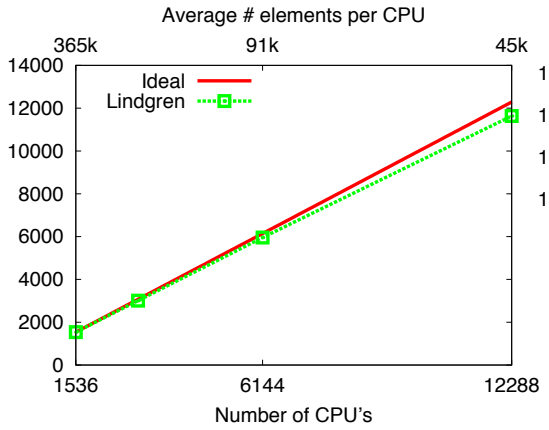
Marenostrum Intel (S): 5000 CPU's compressible flow, incompressible flow, thermal flow (scalability test)

Blue Waters Cray (USA): 130,000 CPU's solid mechanics, electrophysiology, combustion (scalability test)

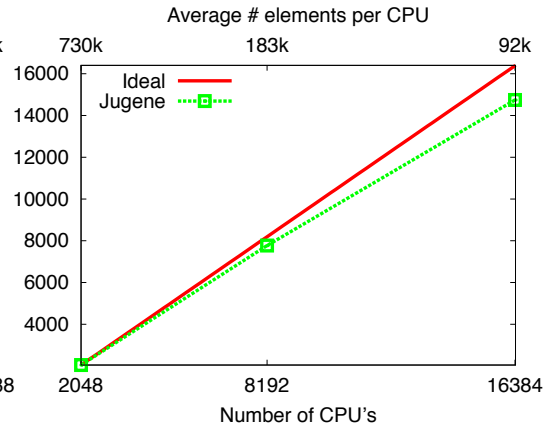
Occigen Bull (F): 20,000 CPUS, Incompressible flow and particle transport in respiratory system (Grand challenge)

# Alya speedup

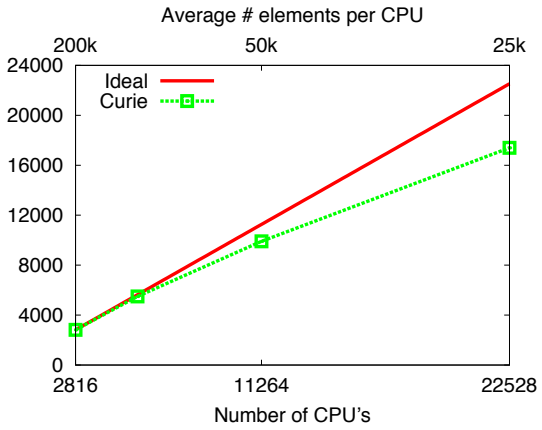
## Lindgren - Cray XE6
## Sweden



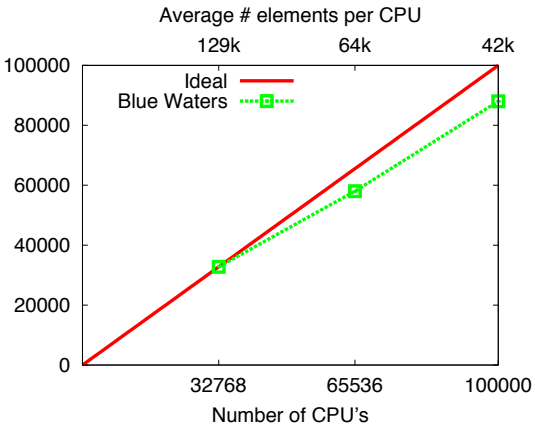

## Jugene - Blue Gene/P
## Germany



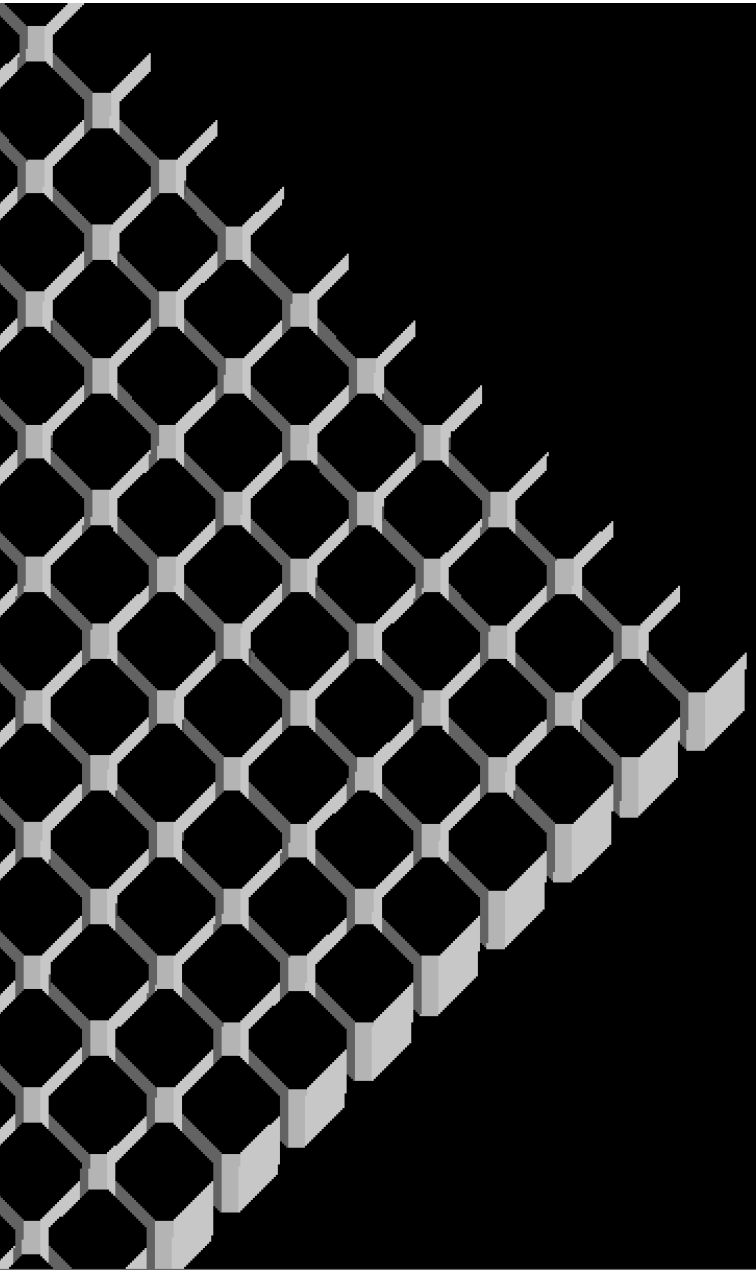

## Curie - BullX
## France



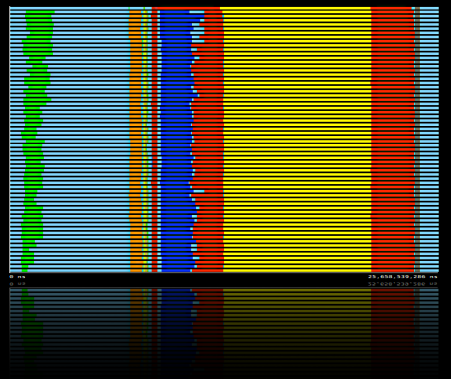

## Blue Waters - Cray XE6
## USA



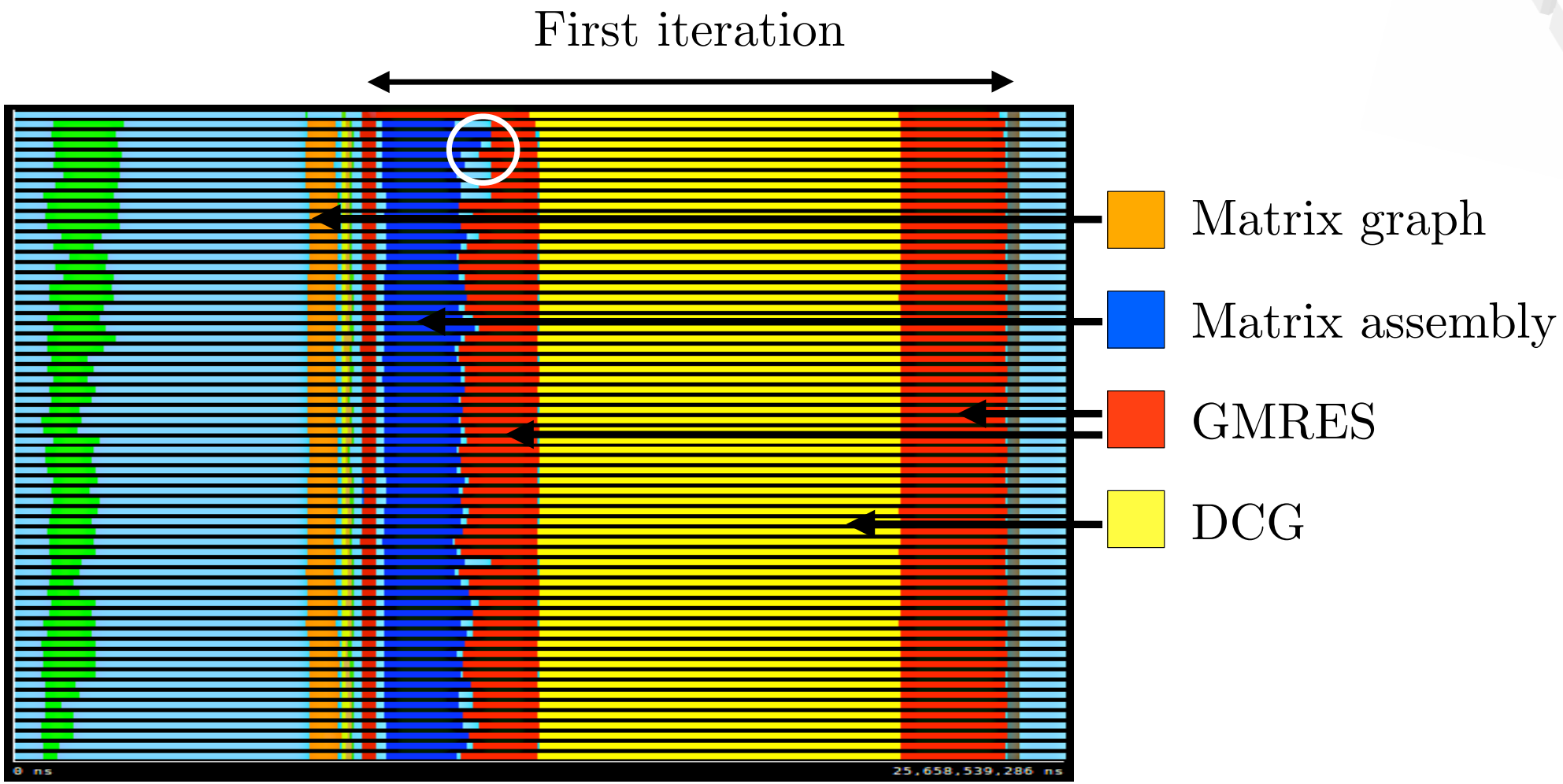


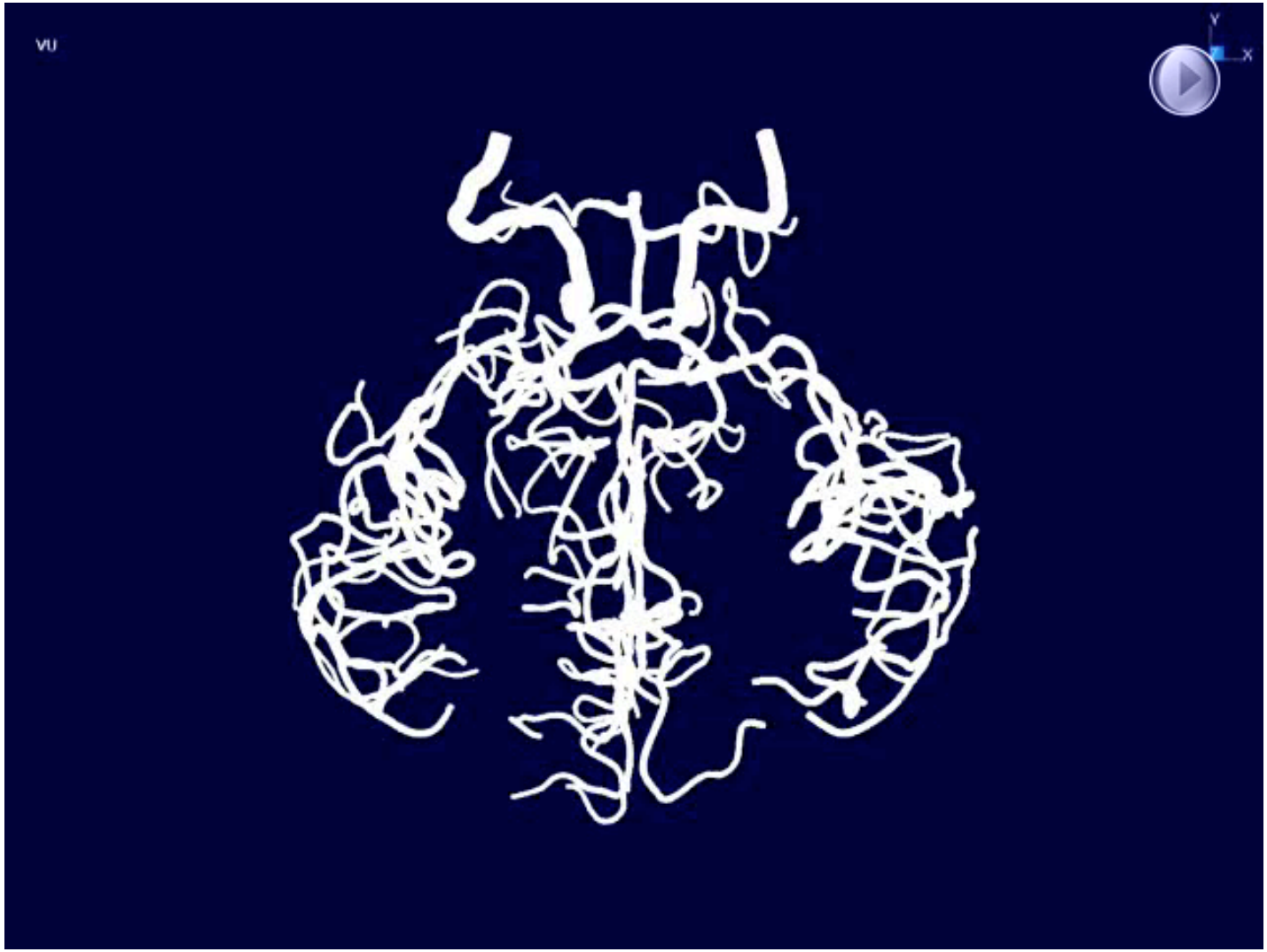
# Performance analysis



# Paraver: performance analysis tool



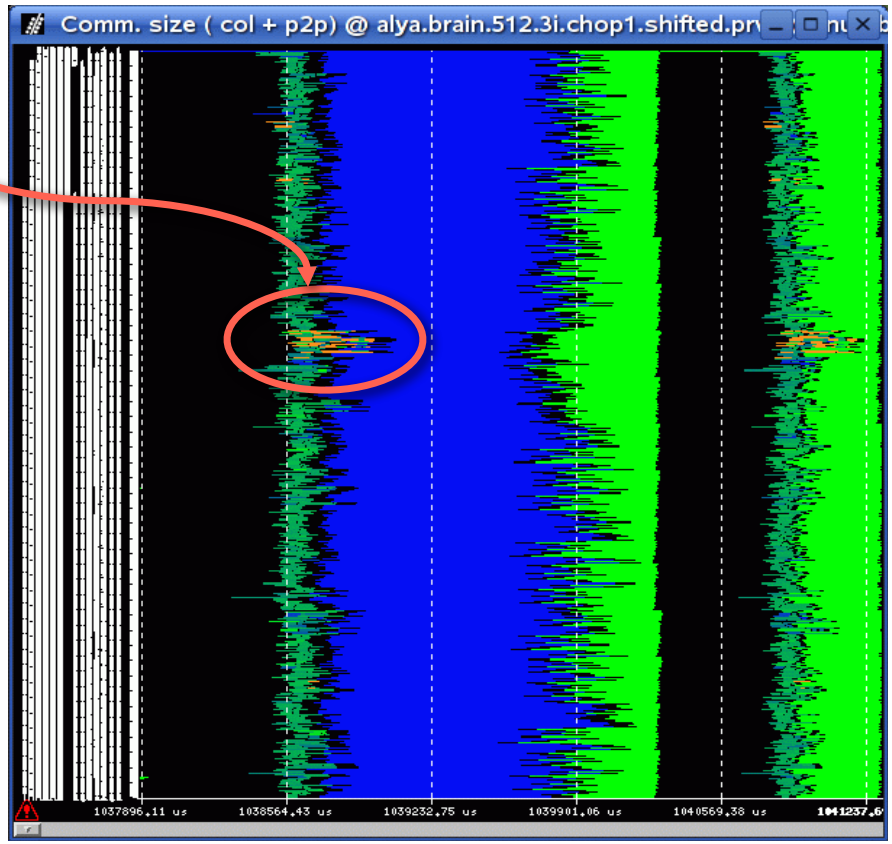
# Identify bottlenecks



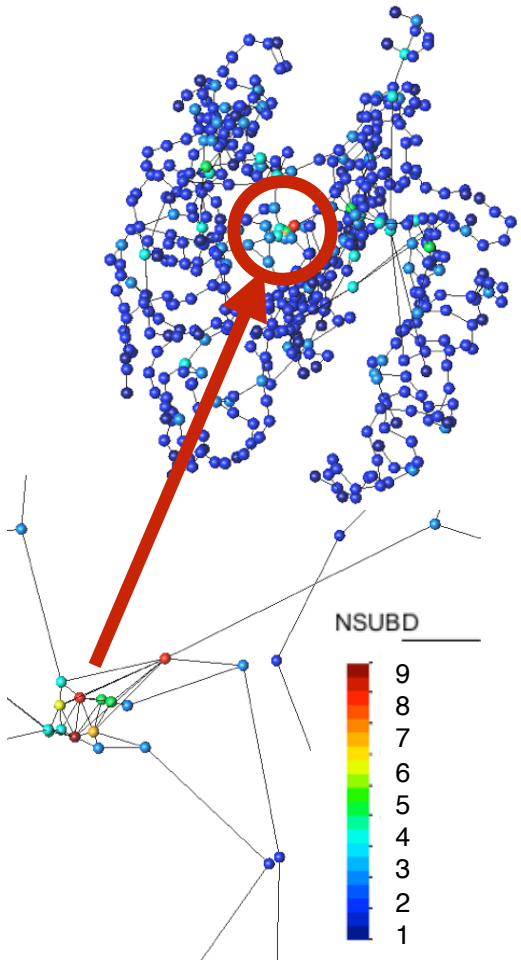
# Identify bottlenecks

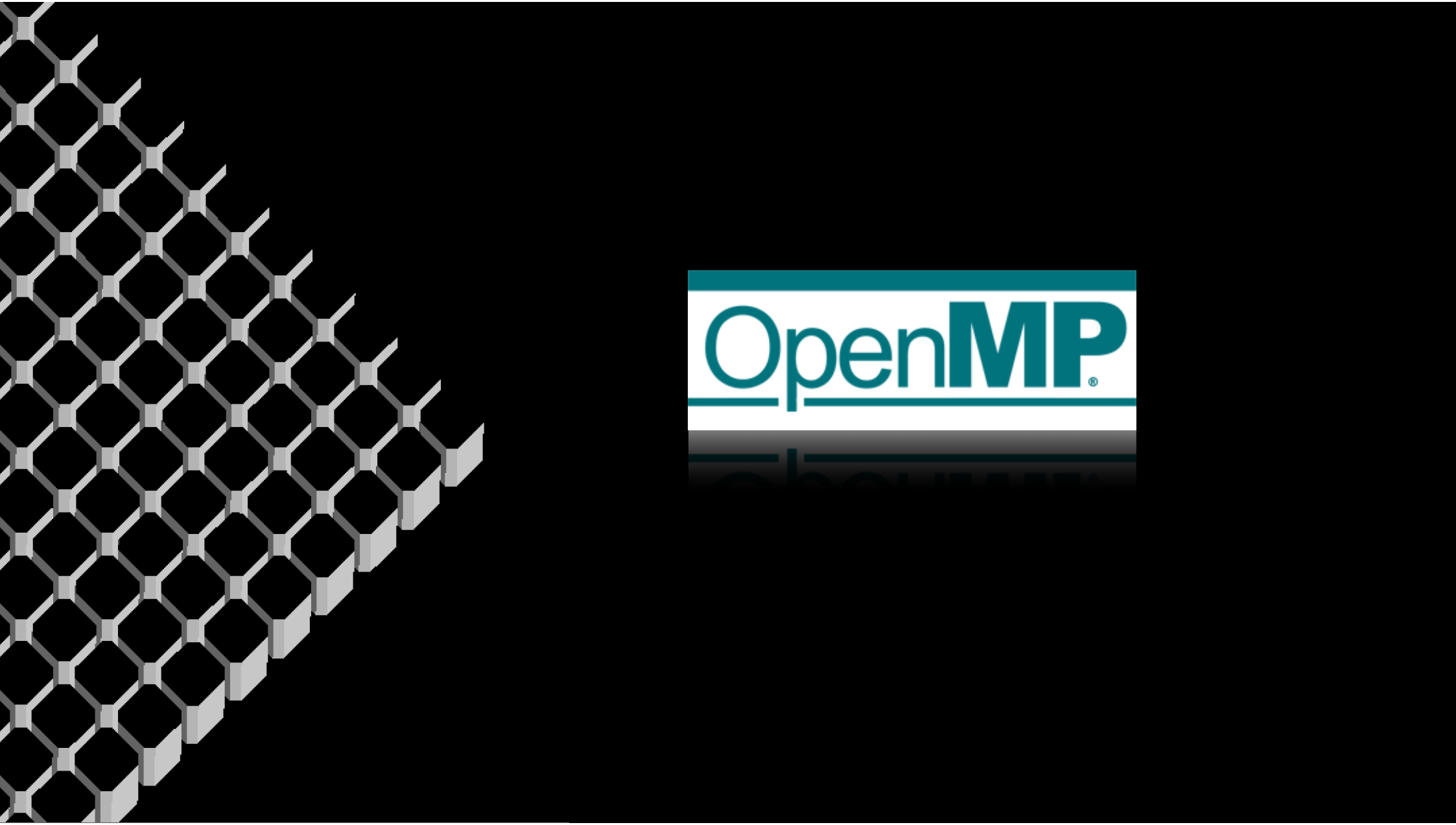
Matrix-vector product | MPI_Allreduce 4000b | Matrix-vector product | MPI_Allreduce 8b

Subdomains with lots of neighbors




MPI_Sendrecv | MPI_Allreduce 8b | MPI_Sendrecv




OpenMP

# What is it?

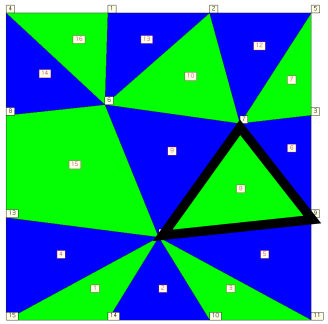
- OpenMP (Open Multi-Processing) is an application programming interface (API).
- It supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran.
- Available on most platforms, processor architectures and operating systems.
- It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.

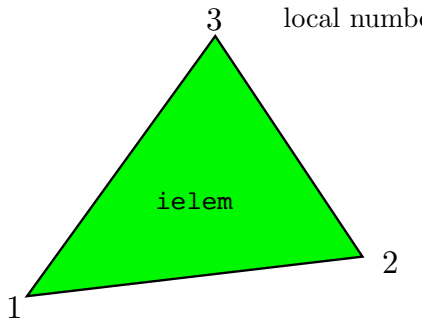
Example: compute $\mathbf{z} = \alpha\mathbf{x} + \mathbf{y}$

```
!$OMP PARALLEL DO SCHEDULE (STATIC)      &
!$OMP DEFAULT  ( NONE )                  &
!$OMP SHARED   ( xx, zz, yy, alpha, nn ) &
!$OMP PRIVATE  ( ii )

do ii = 1,nn
  zz(ii) = alpha * xx(ii) + yy(ii)
end do

!$OMP END PARALLEL DO
```

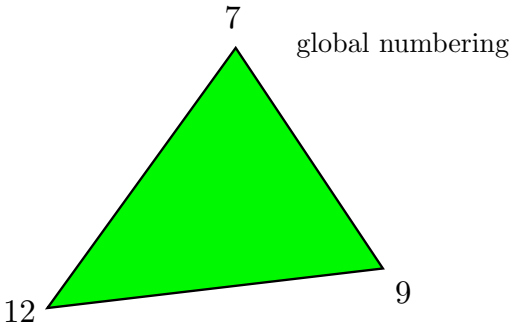
# Assembly

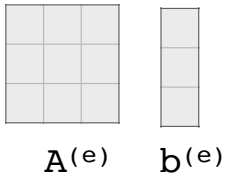
*Compute elemental matrix and RHS and assemble them into the global and matrix and RHS*



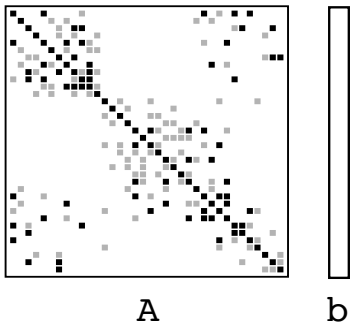



use element connectivity




use element connectivity



# Assembly with OpenMP

```
!$OMP PARALLEL DO SCHEDULE(STATIC) &
!$OMP DEFAULT   (NONE)             &
!$OMP PRIVATE   (…)                &
!$OMP SHARED    (…)                &
!$OMP REDUCTION (+,MIN,MAX:…)

do e = 1,nelem

  ! Element matrix and RHS
   Compute A(e), b(e)

  ! Assembly in global matrix and RHS
  ! OMP CRITICAL
    A <= A + A(e)
    b <= b + b(e)
  ! OMP END CRITICAL

end do

!$OMP END PARALLEL DO
```

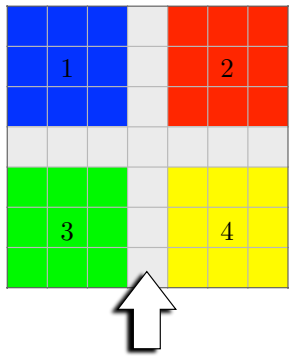
CRITICAL

Two threads access same global position

# Avoiding criticals

## 1. A la MPI

One thread per subdomain




boundary elements

```
!$OMP PARALLEL DO SCHEDULE(STATIC) &
!$OMP PRIVATE    (…)               &
!$OMP SHARED     (…)               &

do i = 1,nsubd
   do ei = 1,nelem_interior(isubd)
      e = list_interior_elements(ei)
      Compute A(e), b(e)
      A <= A + A(e)
      b <= b + b(e)
   end do
end do

!$OMP END PARALLEL DO

!$OMP PARALLEL DO SCHEDULE(STATIC) &
!$OMP PRIVATE    (…)               &
!$OMP SHARED     (…)               &

do eb = 1,nelem_boundary
   e = list_boundary_elements(eb)
   Compute A(e), b(e)
   ! OMP CRITICAL
   A <= A + A(e)
   b <= b + b(e)
   ! OMP END CRITICAL
   end do
end do

!$OMP END PARALLEL DO
```

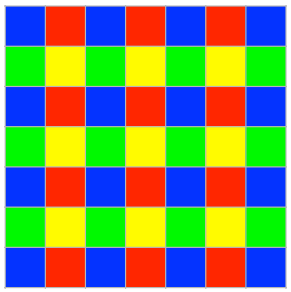
loop over subdomains: NO CRITICAL

loop over boundary elements: CRITICAL

CRITICAL

## 2. Coloring

Distribute element of same color in threads

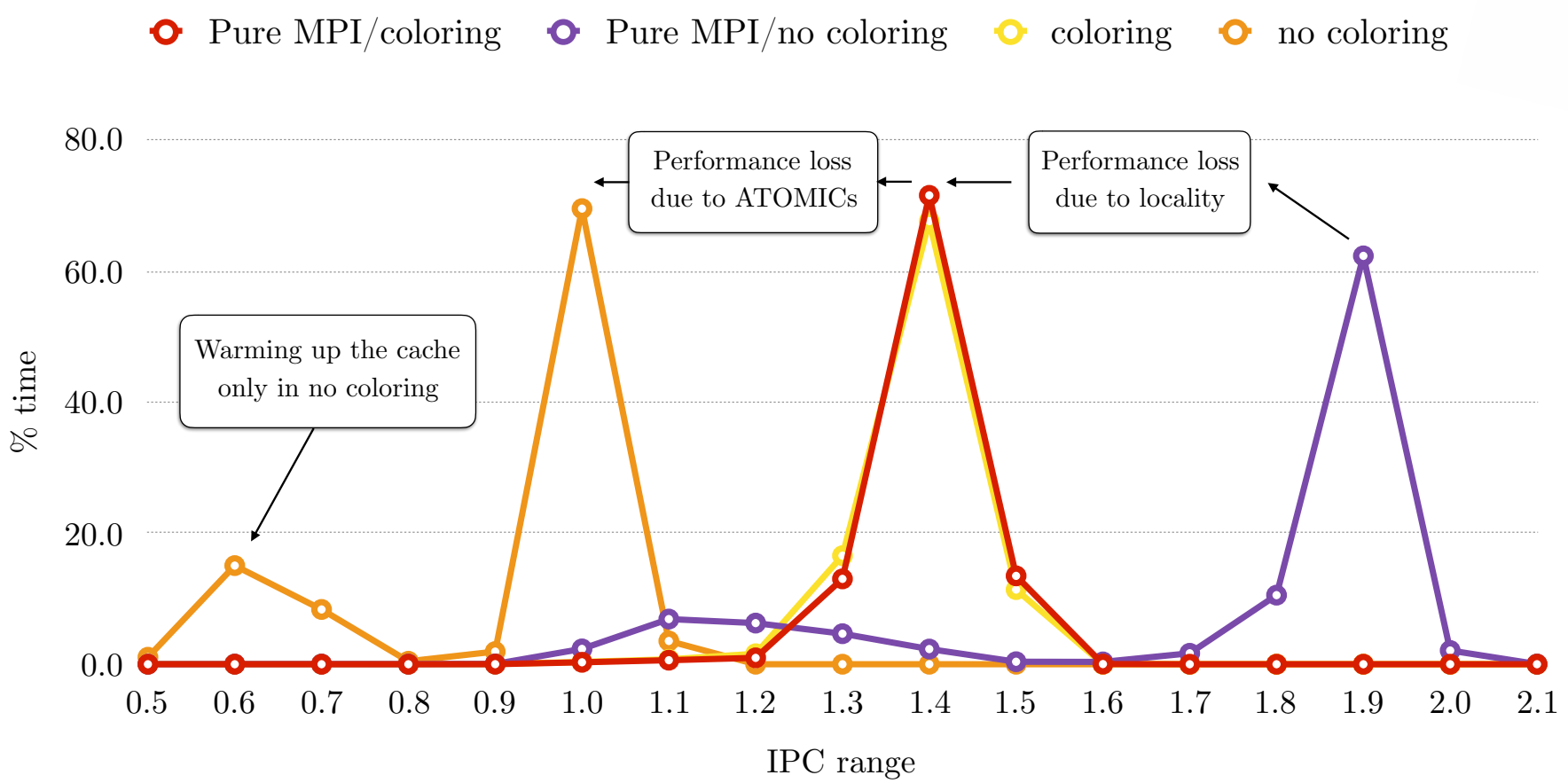


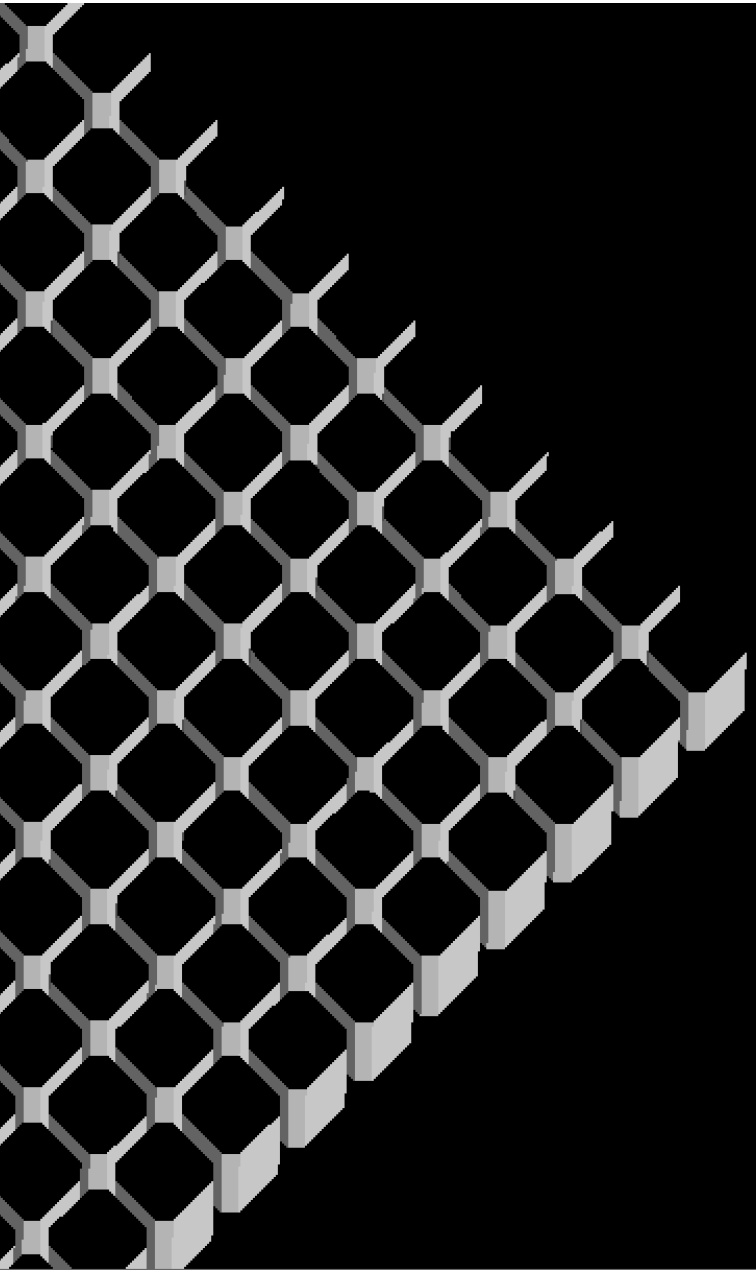
elements of same color do not have nodes in common

```
do icolo = 1,number_colors
   !$OMP PARALLEL DO SCHEDULE(STATIC) &
   !$OMP PRIVATE    (…)               &
   !$OMP SHARED     (…)               &
   do ec = 1,nelem_colors(icolo)
      e = list_color_elements(ec)
      Compute A(e), b(e)
      A <= A + A(e)
      b <= b + b(e)
   end do
end do

!$OMP END PARALLEL DO
```

Chunks may be smaller but we avoid criticals!

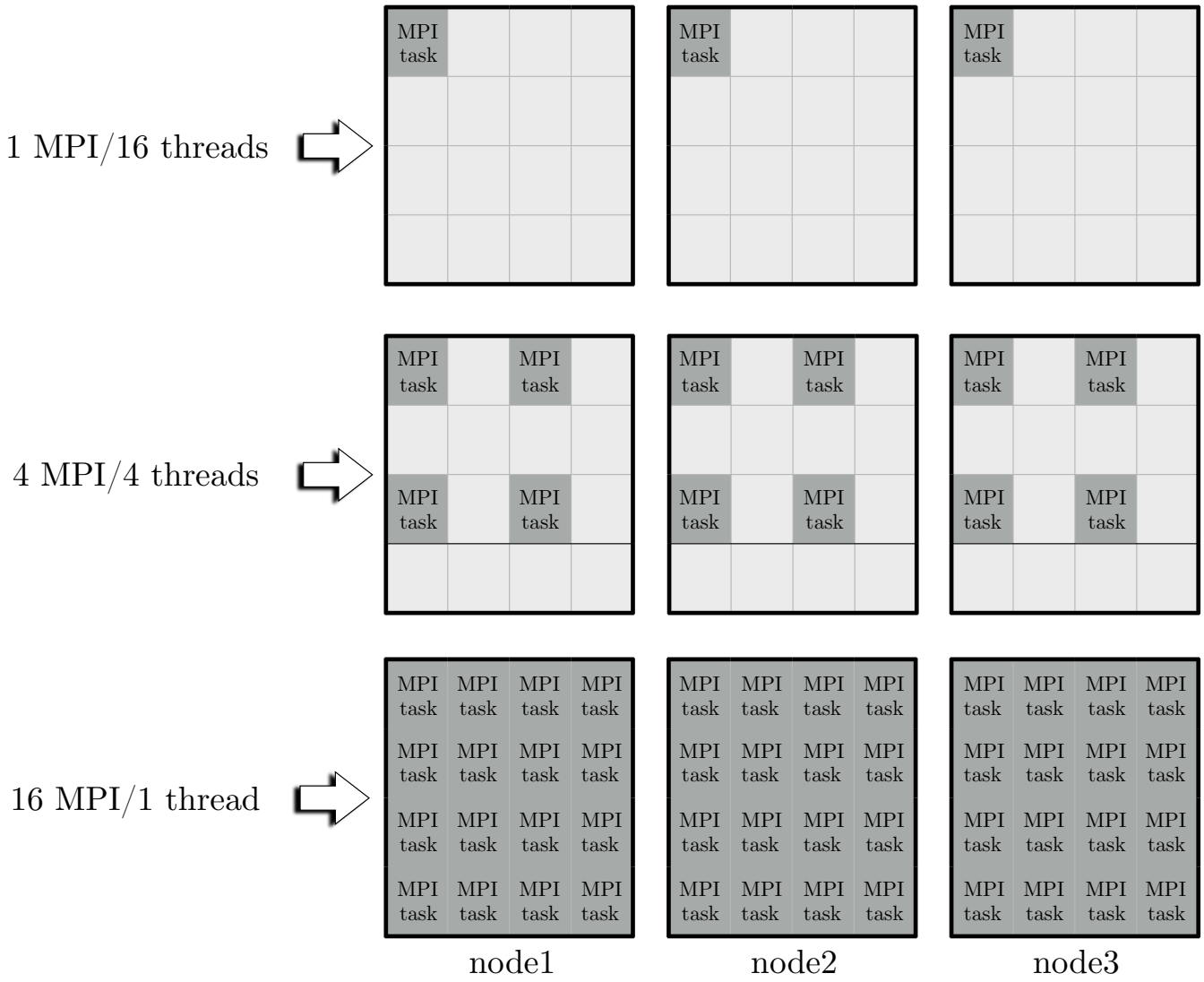
# coloring *vs* no coloring





# Hybrid parallelization

# What is the best configuration?

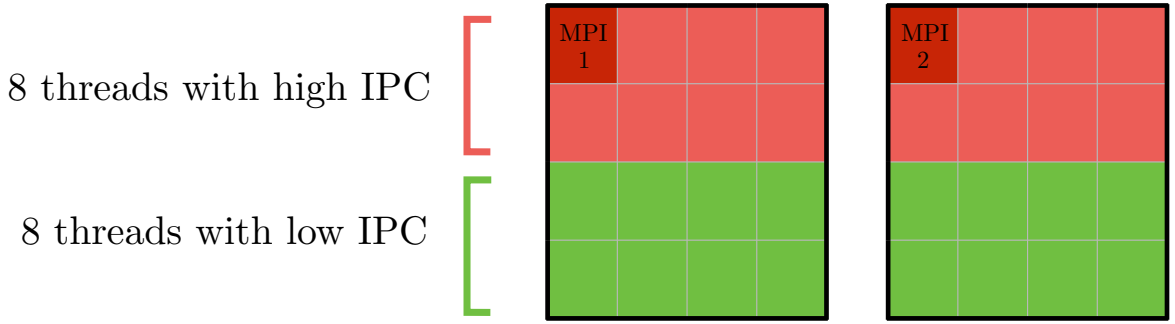


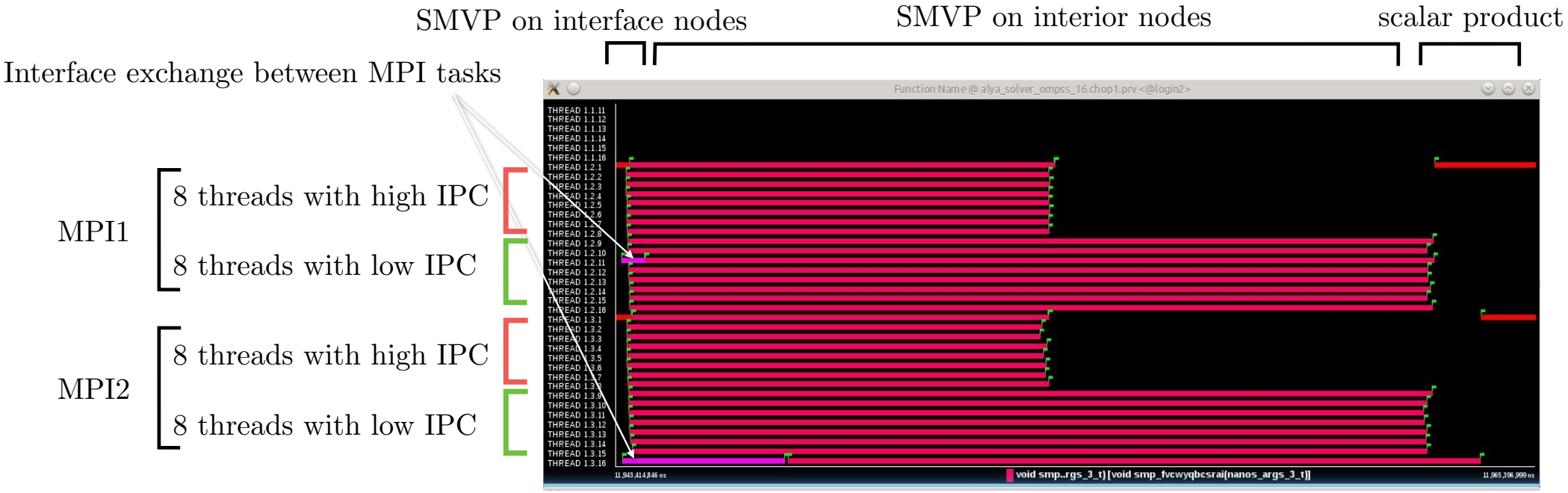
# What is the best configuration?

Criteria to take into account:

- Difficulty to maintain OpenMP
- Overhead OpenMP
- Overhead ATOMIC/CRITICAL
- Worse IPC with coloring
- Load balance worse as number of MPI increases
- Which loops should be parallelized
- Configuration of the node

# Example of *very* bad configuration

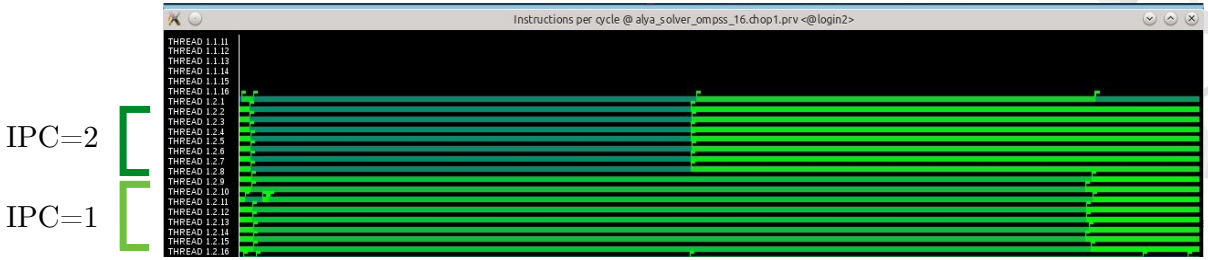


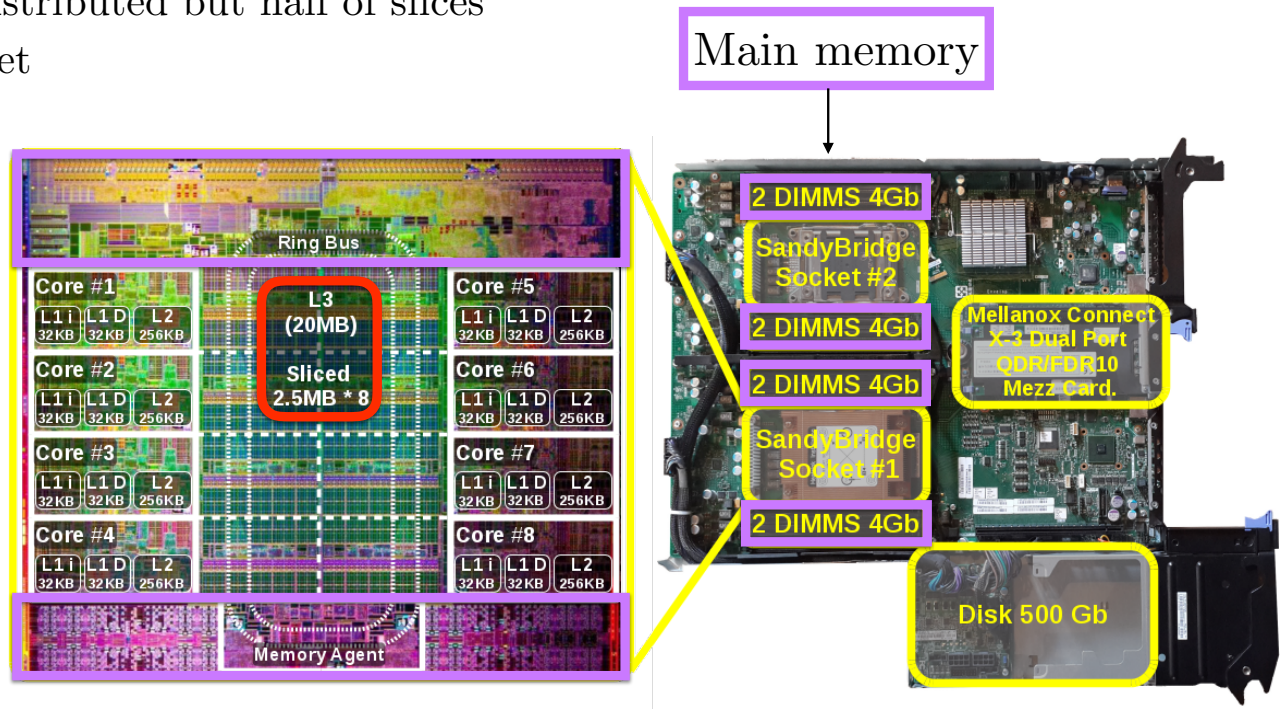



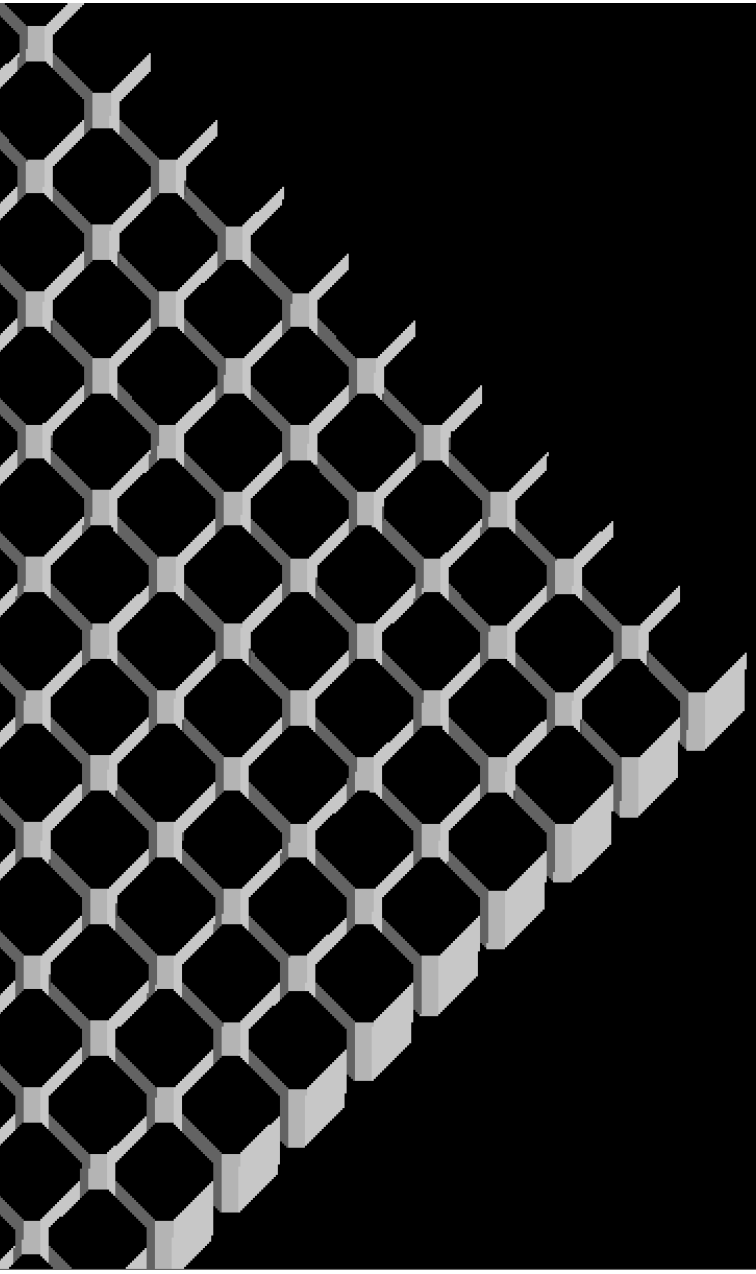
# Example of *very* bad configuration

**1 MareNostrum node**

- 2 sockets with 8 cores each
- L1, L2 local to each core
- L3 is local to socket (shared by 8 cores)
- Each L3 slice is next to one core
- Main memory distributed but half of slices next to one socket



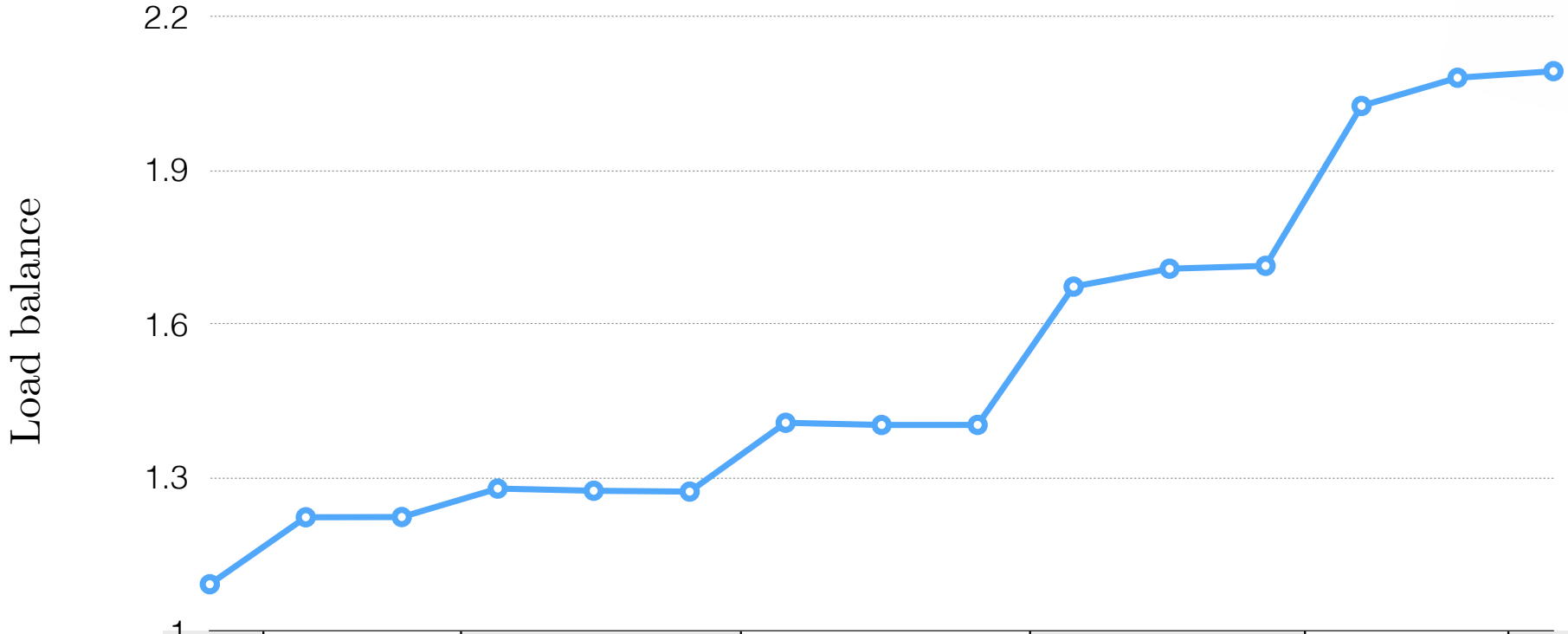


# Introducing dynamic load balance

# Dynamic load balance

Load balancing worsens with number of subdomains

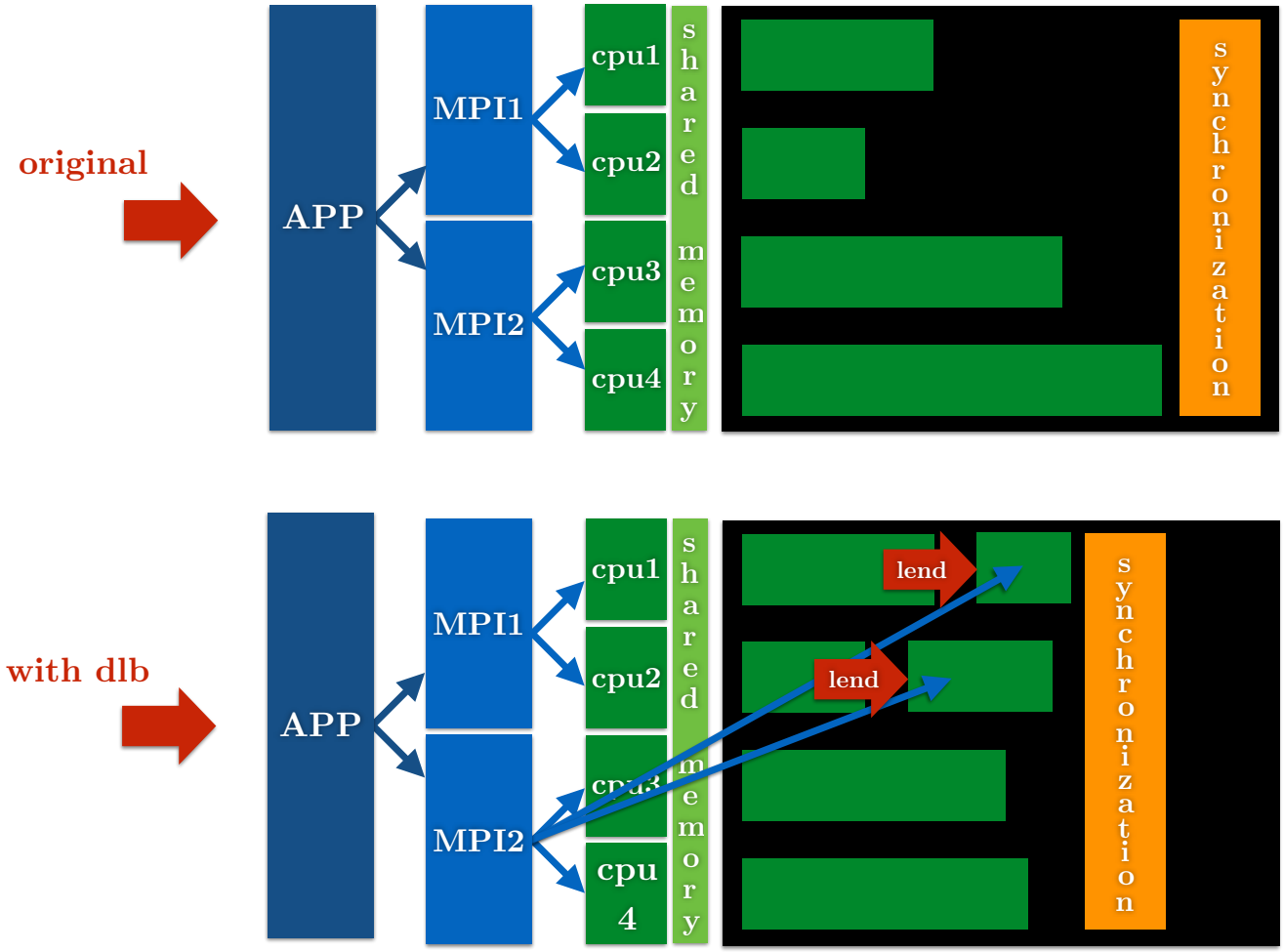
Load balance=*Max time/Ave time*




| $n_{\mathrm{MPI/node}}$ x $n_{\mathrm{OMP/MPI}}$ | 1x16 | 2x8 | 1x16 | 4x4 | 2x8 | 1x16 | 8x2 | 4x4 | 2x8 | 16x1 | 8x2 | 4x4 | 16x1 | 8x2 | 16x1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $n_{\mathrm{MPI}}$ | 16 | 32 | | 64 | | | 128 | | | 256 | | | 512 | | 1024 |

# Dynamic load balance

Idea: share resources between the different MPI tasks




dynamic load balancing

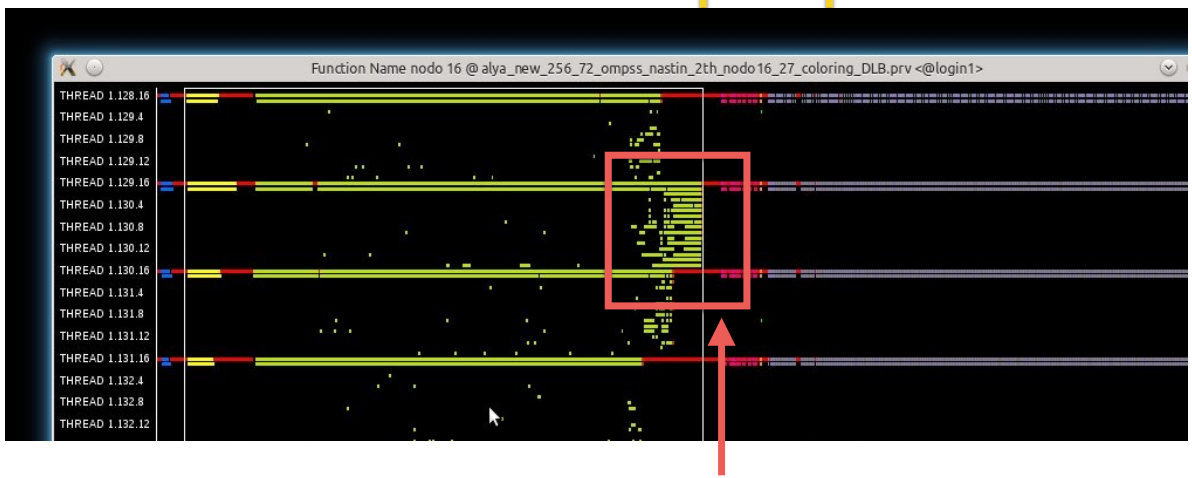
# Dynamic load balance

Without dlb


MPI tasks with more work

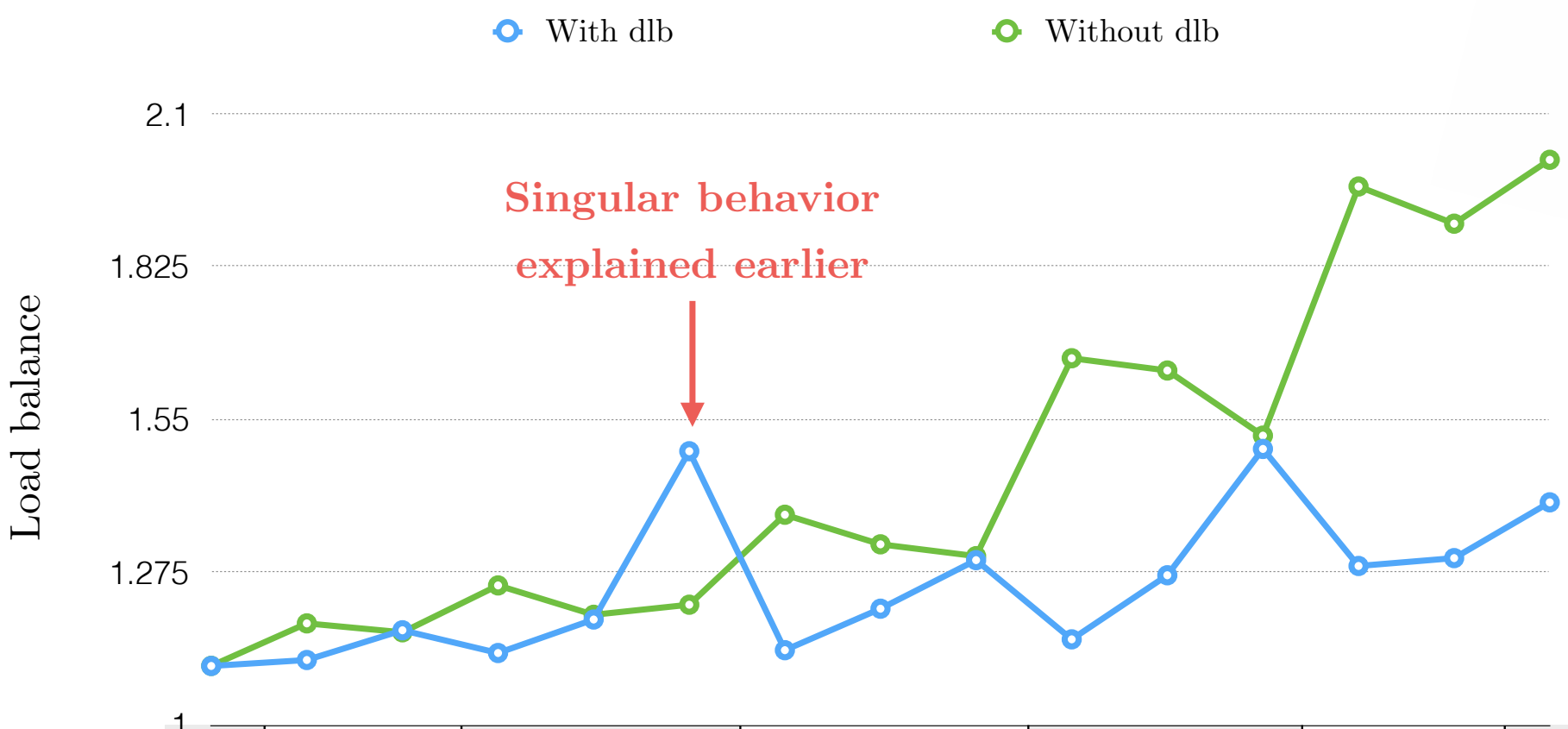
Time reduction

With dlb



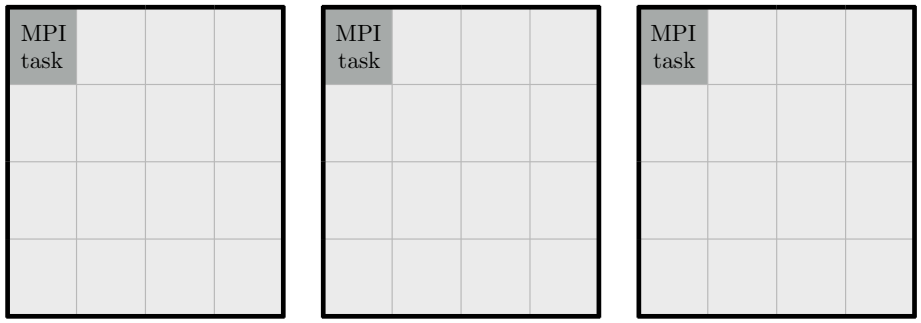

This MPI tasks uses the resources of the others on the node

# Dynamic load balance




| $n_{\text{MPI/node}}$ x $n_{\text{OMP/MPI}}$ | 1x16 | 2x8 | 1x16 | 4x4 | 2x8 | 1x16 | 8x2 | 4x4 | 2x8 | 16x1 | 8x2 | 4x4 | 16x1 | 8x2 | 16x1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $n_{\text{MPI}}$ | 16 | 32 | | 64 | | | 128 | | | 256 | | | 512 | | 1024 |

# Best configuration?

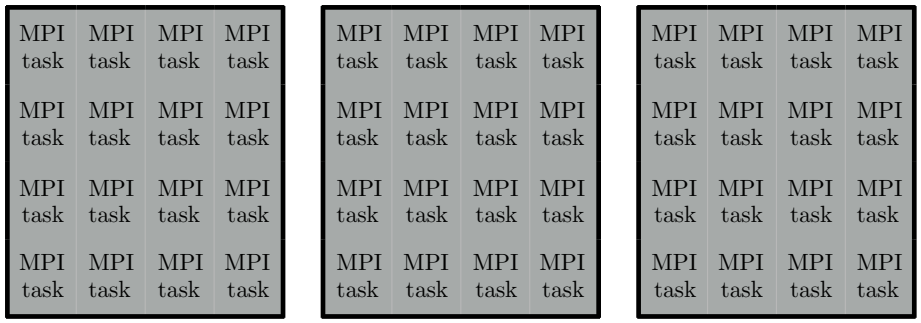



No, because dynamic load balance is carried out at the node level!

So we have no way to reduce the load imbalance

# Best configuration?

Fill the nodes with MPI is the best choice?

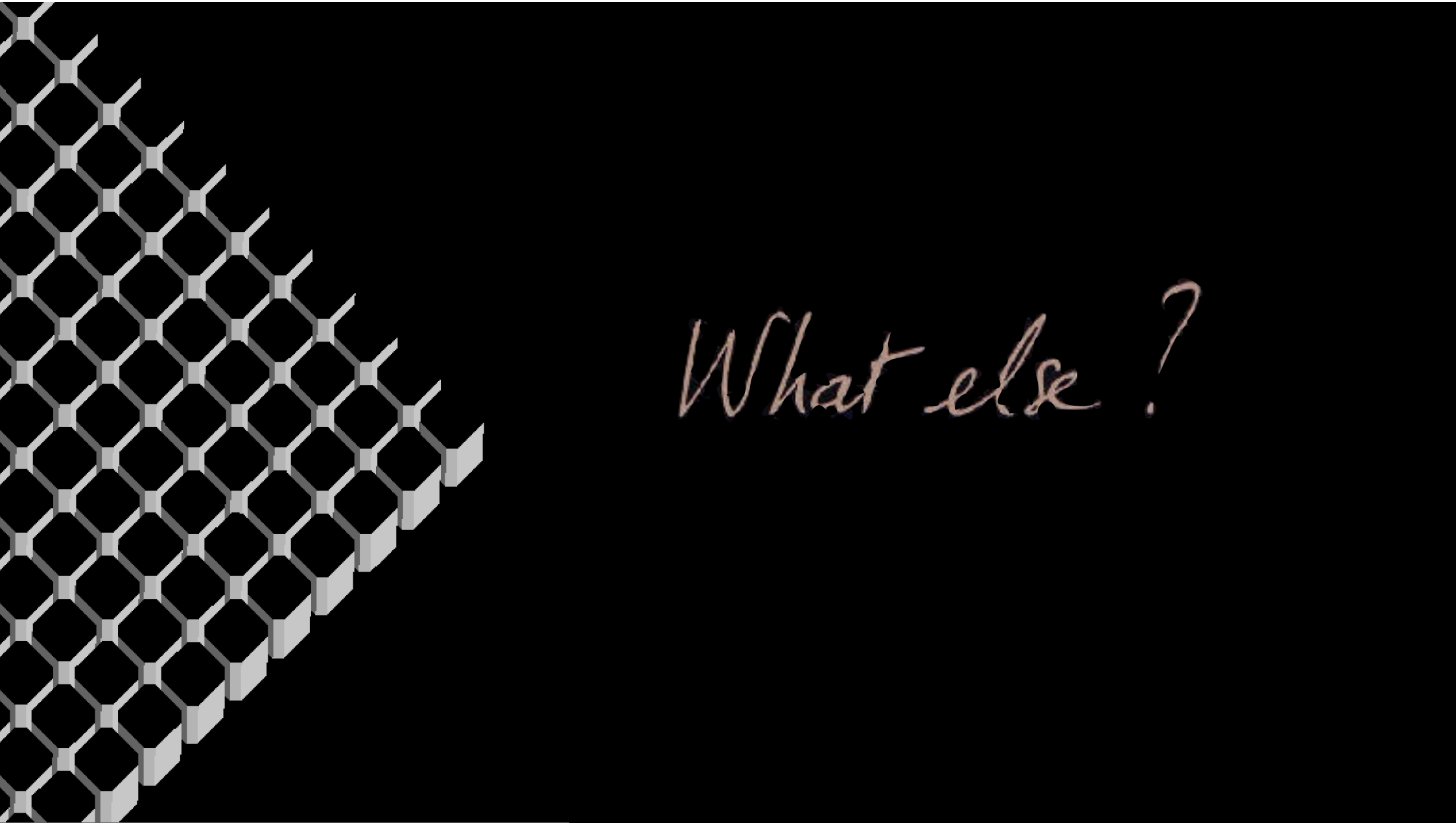



PROs

- Let dlb load balance
- No need to parallelize all the code with openMP
- Only put OpenMP where load imbalance is likely to occur

CONTRAs

- Load imbalance increases with number of subdomains


What else ?

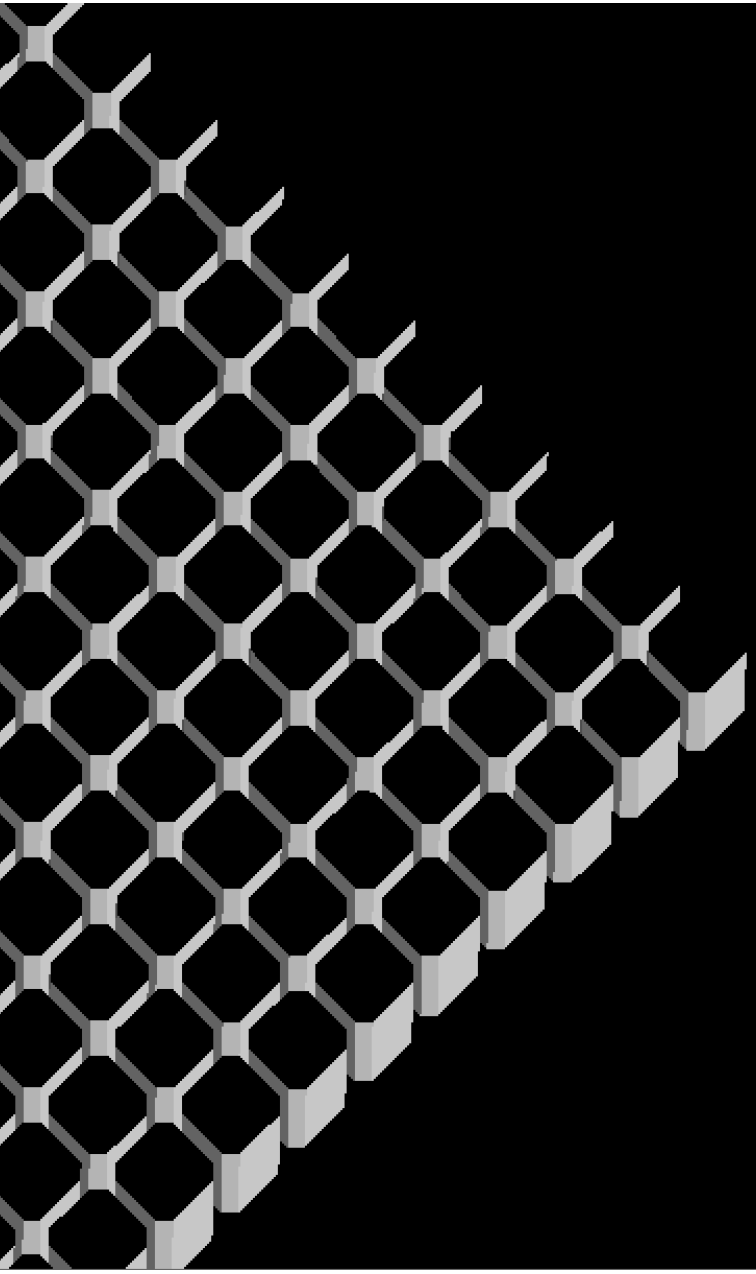
# What else?

We have only went through the basics

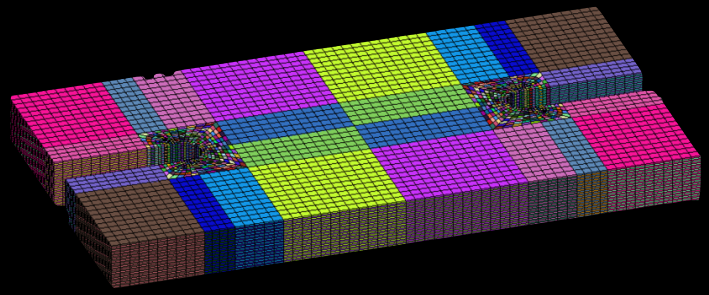
- Matrix-vector product
- Scalar product

! A parallel code is full of different calls to MPI

- Exchange between element faces (crack propagation)
- Second neighboring node exchange (geometrical computations)
- Embedded mesh (see Samaniego's talk)
- Multi-physics multi-domain coupling (see next talk)
- Full interface stencil (Schwarz, Schur complement solver)...



# Pipelined Conjugate Gradient



# Pipelined Conjugate Gradient (PCG)

**Algorithm 1** Preconditioned CG

$r_0 := b - Ax_0; \quad u_0 := M^{-1}r_0; \quad p_0 := u_0$
**for** $i = 0, \ldots$ **do**
  $s := Ap_i$
  $\alpha := (r_i, u_i) \,/\, (s, p_i)$
  $x_{i+1} := x_i + \alpha p_i$
  $r_{i+1} := r_i - \alpha s$
  $u_{i+1} := M^{-1}r_{i+1}$
  $\beta := (r_{i+1}, u_{i+1}) \,/\, (r_i, u_i)$
  $p_{i+1} := u_{i+1} + \beta p_i$
**end for**

- Scalar products are synchronization points
- Introduce more variables to move and join the scalar products
- Use MPI3 to overlap communications and computation

# Pipelined Conjugate Gradient (PCG)

**Algorithm 1** Preconditioned CG

$r_0 := b - Ax_0; \quad u_0 := M^{-1}r_0; \quad p_0 := u_0$
**for** $i = 0, \ldots$ **do**
  $s := Ap_i$
  $\alpha := (r_i, u_i) \,/\, (s, p_i)$
  $x_{i+1} := x_i + \alpha p_i$
  $r_{i+1} := r_i - \alpha s$
  $u_{i+1} := M^{-1}r_{i+1}$
  $\beta := (r_{i+1}, u_{i+1}) \,/\, (r_i, u_i)$
  $p_{i+1} := u_{i+1} + \beta p_i$
**end for**

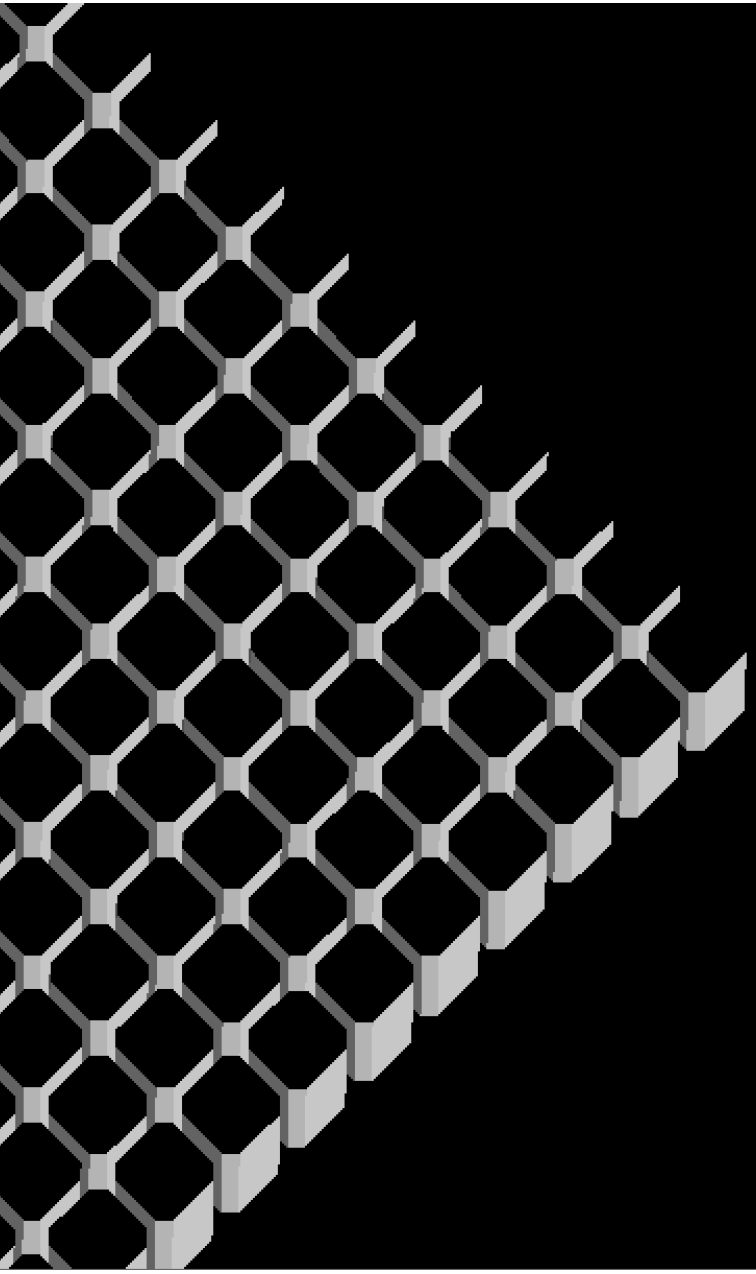
MPI reduction overlaps preconditioning

MPI_Waitall

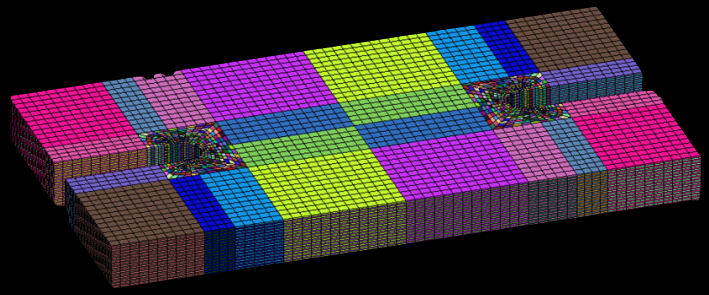
introduce more variables and more operations

**Algorithm 4** Preconditioned pipelined CG

$r_0 := b - Ax_0; \quad u_0 := M^{-1}r_0; \quad w_0 := Au_0$
**for** $i = 0, \ldots$ **do**
  $\gamma_i := (r_i, u_i)$
  $\delta := (w_i, u_i)$
  $m_i := M^{-1}w_i$
  $n_i := Am_i$
  **if** $i > 0$ **then**
    $\beta_i := \gamma_i / \gamma_{i-1}; \quad \alpha_i := \gamma_i / \left(\delta - \beta_i \gamma_i / \alpha_{i-1}\right)$
  **else**
    $\beta_i := 0; \quad \alpha_i := \gamma_i / \delta$
  **end if**
  $z_i := n_i + \beta_i z_{i-1}$
  $q_i := m_i + \beta_i q_{i-1}$
  $s_i := w_i + \beta_i s_{i-1}$
  $p_i := u_i + \beta_i p_{i-1}$
  $x_{i+1} := x_i + \alpha_i p_i$
  $r_{i+1} := r_i - \alpha_i s_i$
  $u_{i+1} := u_i - \alpha_i q_i$
  $w_{i+1} := w_i - \alpha_i z_i$
**end for**

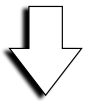


# Deflated Conjugate Gradient



# Deflated Conjugate Gradient (DCG)

Same idea as coarse grain preconditioner



to solve a coarse problem to damp low frequencies quickly

# Deflated Conjugate Gradient (DCG)

## Classical Conjugate Gradient

- Define a preconditioning matrix: $\mathbf{M}$

- Start: Given $\mathbf{x}_{-1}$
  $\mathbf{r}_{-1} = \mathbf{b} - \mathbf{A} \cdot \mathbf{x}_{-1}$

  $\mathbf{x}_0 = \mathbf{x}_{-1}$
  $\mathbf{r}_0 = \mathbf{b} - \mathbf{A} \cdot \mathbf{x}_0$
- Compute: $\mathbf{M} \cdot \mathbf{z}_0 = \mathbf{r}_0$

- Set: $\mathbf{p}_0 =$ $\mathbf{z}_0$
- Do until convergence:
  $\alpha_j = (\mathbf{r}_j \cdot \mathbf{z}_j)/(\mathbf{p}_j \cdot \mathbf{A} \cdot \mathbf{p}_j)$
  $\mathbf{x}_{j+1} = \mathbf{x}_j + \alpha_j \mathbf{p}_j$
  $\mathbf{r}_{j+1} = \mathbf{r}_j - \alpha_j \mathbf{A} \cdot \mathbf{p}_j$
  $\mathbf{M} \cdot \mathbf{z}_{j+1} = \mathbf{r}_{j+1}$
  $\beta_j = (\mathbf{r}_{j+1} \cdot \mathbf{z}_{j+1})/(\mathbf{r}_j \cdot \mathbf{z}_j)$

  $\mathbf{p}_{j+1} = \mathbf{z}_{j+1} + \beta_j \mathbf{p}_j$

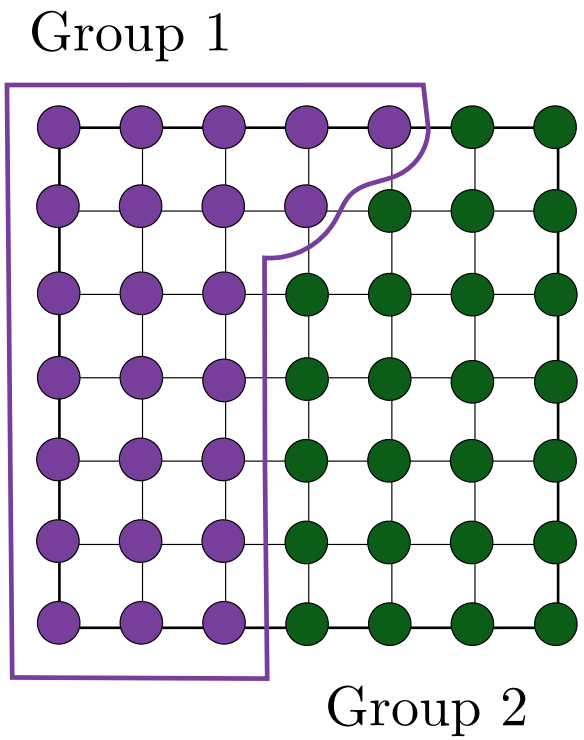
## Deflated Conjugate Gradient

- Define a preconditioning matrix: $\mathbf{M}$
- Define: $\hat{\mathbf{A}} = \mathbf{W}^T \cdot \mathbf{A} \cdot \mathbf{W}$
- Start: Given $\mathbf{x}_{-1}$
  $\mathbf{r}_{-1} = \mathbf{b} - \mathbf{A} \cdot \mathbf{x}_{-1}$
  $\hat{\mathbf{A}} \cdot \mathbf{d}_0 = \mathbf{W}^T \cdot \mathbf{r}_{-1}$
  $\mathbf{x}_0 = \mathbf{x}_{-1} + \mathbf{W} \cdot \mathbf{d}_0$
  $\mathbf{r}_0 = \mathbf{b} - \mathbf{A} \cdot \mathbf{x}_0$
- Compute: $\mathbf{M} \cdot \mathbf{z}_0 = \mathbf{r}_0$
- Solve: $\hat{\mathbf{A}} \cdot \mathbf{d} = \mathbf{W}^T \cdot \mathbf{A} \cdot \mathbf{z}_0$
- Set: $\mathbf{p}_0 = -\mathbf{W} \cdot \mathbf{d} + \mathbf{z}_0$
- Do until convergence:
  $\alpha_j = (\mathbf{r}_j \cdot \mathbf{z}_j)/(\mathbf{p}_j \cdot \mathbf{A} \cdot \mathbf{p}_j)$
  $\mathbf{x}_{j+1} = \mathbf{x}_j + \alpha_j \mathbf{p}_j$
  $\mathbf{r}_{j+1} = \mathbf{r}_j - \alpha_j \mathbf{A} \cdot \mathbf{p}_j$
  $\mathbf{M} \cdot \mathbf{z}_{j+1} = \mathbf{r}_{j+1}$
  $\beta_j = (\mathbf{r}_{j+1} \cdot \mathbf{z}_{j+1})/(\mathbf{r}_j \cdot \mathbf{z}_j)$
  $\hat{\mathbf{A}} \cdot \mathbf{d}_j = \mathbf{W}^T \cdot \mathbf{A} \cdot \mathbf{z}_{j+1}$
  $\mathbf{p}_{j+1} = \mathbf{z}_{j+1} + \beta_j \mathbf{p}_j - \mathbf{W} \cdot \mathbf{d}_j$

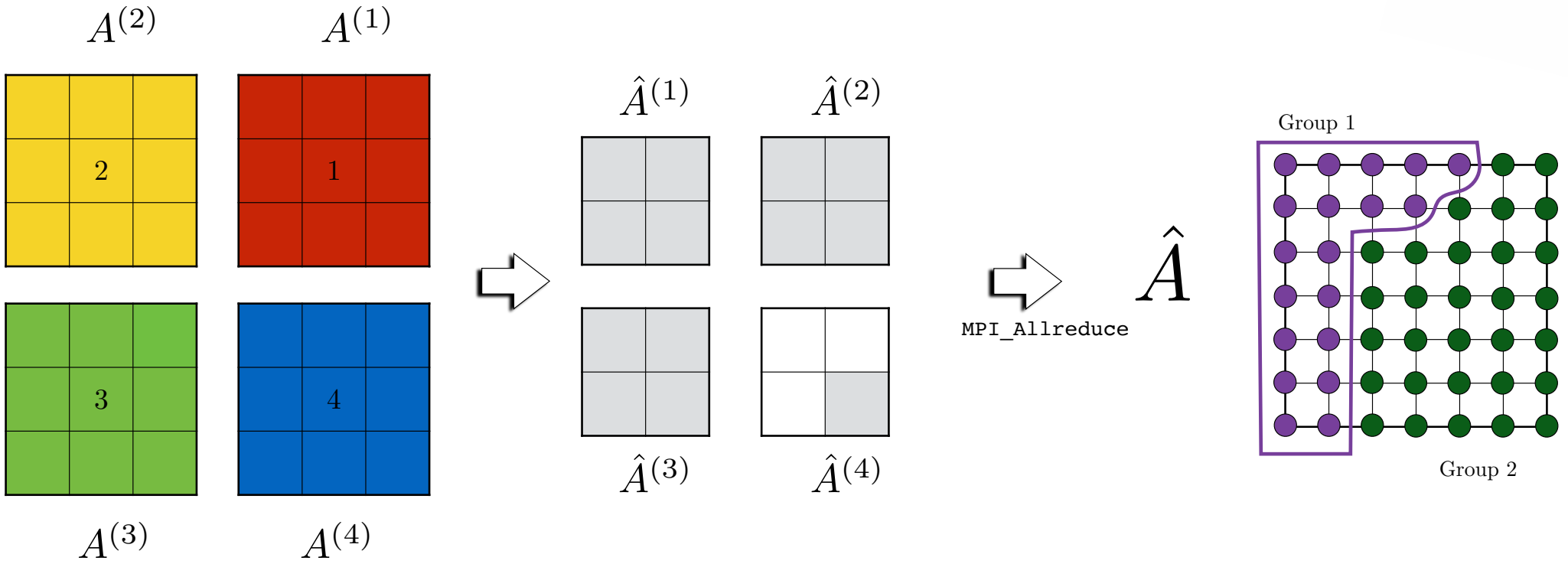
# Deflated Conjugate Gradient (DCG)

Group nodes into groups

Example: 2 groups

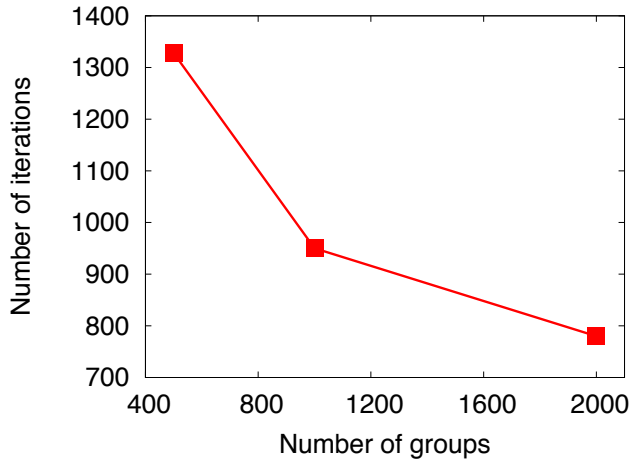


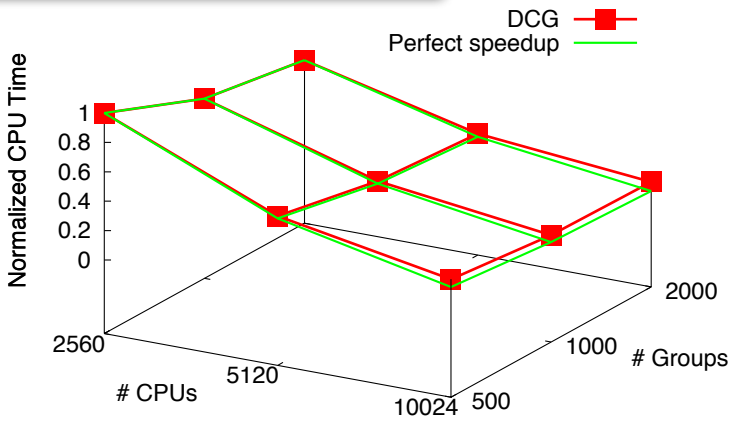
# Deflated Conjugate Gradient (DCG)

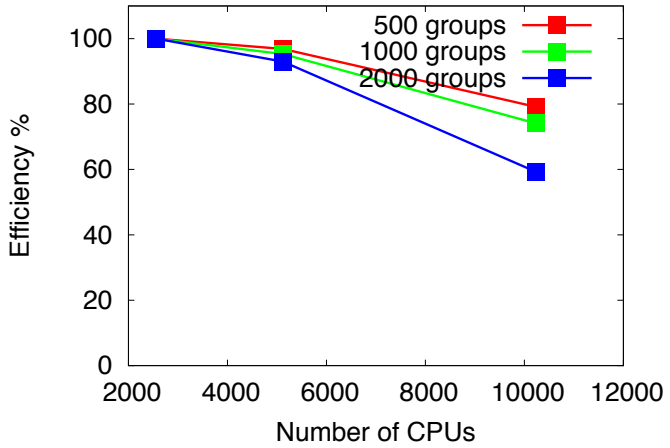


# Deflated Conjugate Gradient (DCG)

DCG - 553M mesh - 2560 -> 10240 CPUs



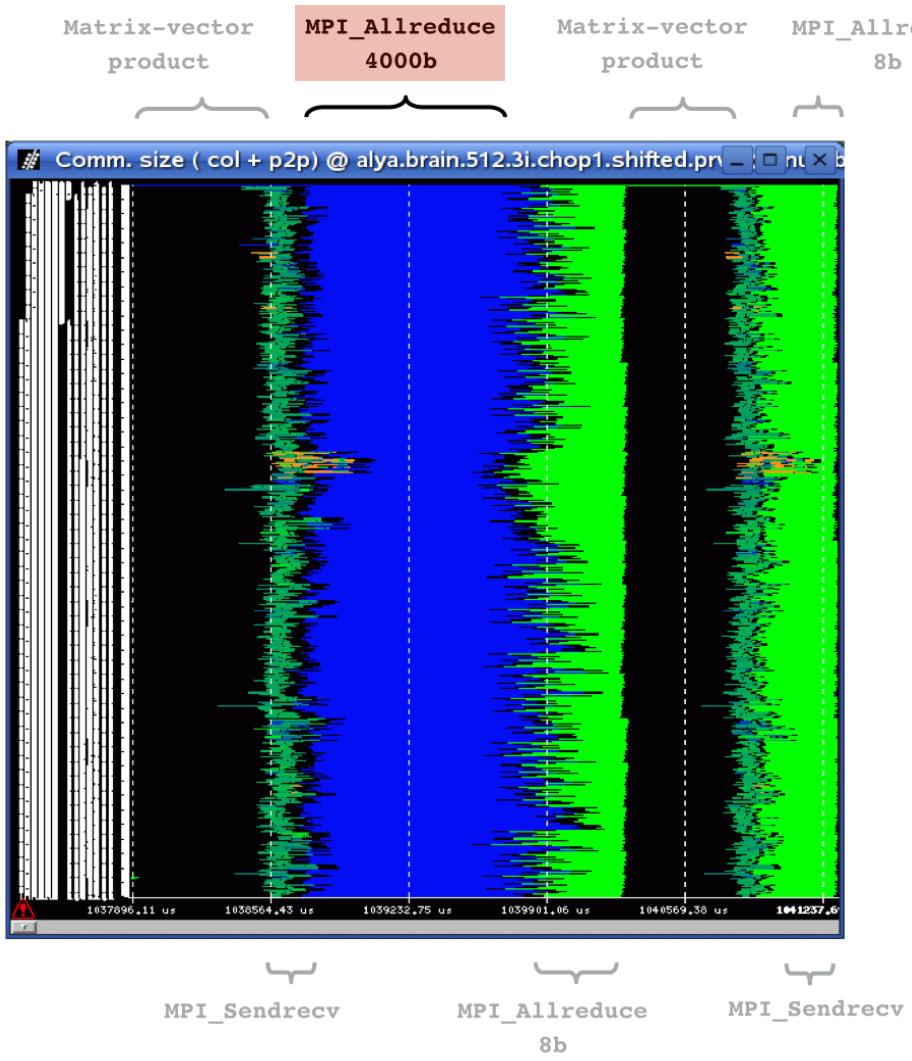



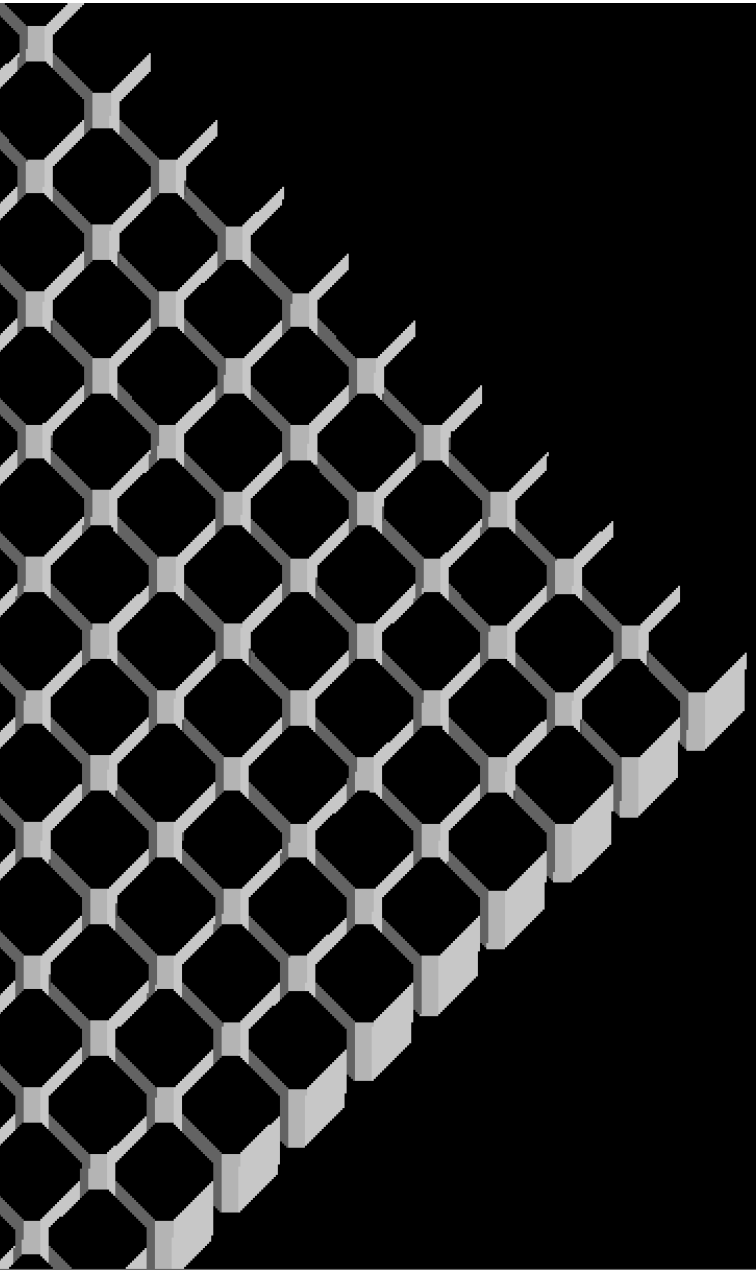

When # groups increases:

- ✔ Convergence is better BUT:
- ✖ Cost of $\hat{A}^{-1}$ increases
- ✖ Communications increases

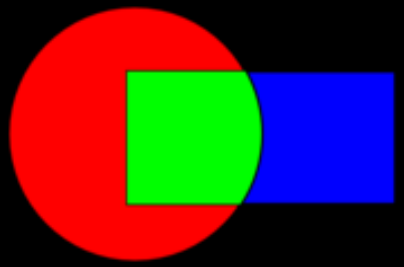
# Deflated Conjugate Gradient (DCG)




- Define a preconditioning matrix: $\mathbf{M}$
- Define: $\hat{\mathbf{A}} = \mathbf{W}^T \cdot \mathbf{A} \cdot \mathbf{W}$
- Start: Given $\mathbf{x}_{-1}$
  $\mathbf{r}_{-1} = \mathbf{b} - \mathbf{A} \cdot \mathbf{x}_{-1}$
  $\hat{\mathbf{A}} \cdot \mathbf{d}_0 = \mathbf{W}^T \cdot \mathbf{r}_{-1}$
  $\mathbf{x}_0 = \mathbf{x}_{-1} + \mathbf{W} \cdot \mathbf{d}_0$
  $\mathbf{r}_0 = \mathbf{b} - \mathbf{A} \cdot \mathbf{x}_0$
- Compute: $\mathbf{M} \cdot \mathbf{z}_0 = \mathbf{r}_0$
- Solve: $\hat{\mathbf{A}} \cdot \mathbf{d} = \mathbf{W}^T \cdot \mathbf{A} \cdot \mathbf{z}_0$
- Set: $\mathbf{p}_0 = -\mathbf{W} \cdot \mathbf{d} + \mathbf{z}_0$
- Do until convergence:
  $\alpha_j = (\mathbf{r}_j \cdot \mathbf{z}_j)/(\mathbf{p}_j \cdot \mathbf{A} \cdot \mathbf{p}_j)$
  $\mathbf{x}_{j+1} = \mathbf{x}_j + \alpha_j \mathbf{p}_j$
  $\mathbf{r}_{j+1} = \mathbf{r}_j - \alpha_j \mathbf{A} \cdot \mathbf{p}_j$
  $\mathbf{M} \cdot \mathbf{z}_{j+1} = \mathbf{r}_{j+1}$
  $\beta_j = (\mathbf{r}_{j+1} \cdot \mathbf{z}_{j+1})/(\mathbf{r}_j \cdot \mathbf{z}_j)$
  $\hat{\mathbf{A}} \cdot \mathbf{d}_j = \mathbf{W}^T \cdot \mathbf{A} \cdot \mathbf{z}_{j+1}$
  $\mathbf{p}_{j+1} = \mathbf{z}_{j+1} + \beta_j \mathbf{p}_j - \mathbf{W} \cdot \mathbf{d}_j$



# Restricted Additive Schwarz



# In addition to...

Not everything is

scalar product

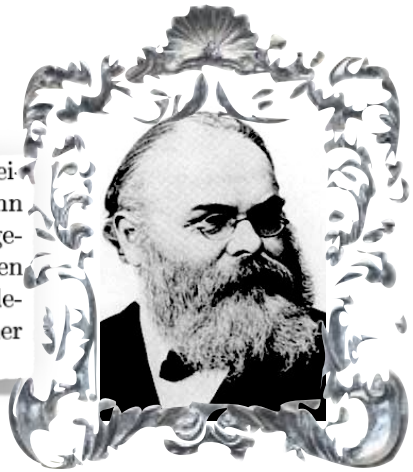
and

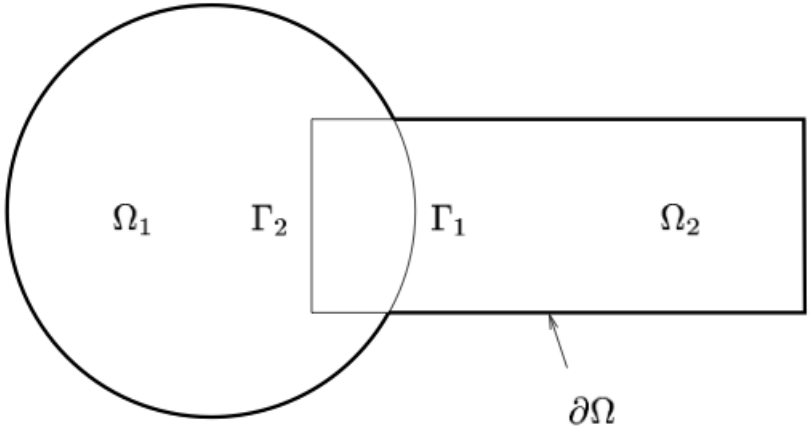
matrix-vector product

# Schwarz

*What is Schwarz?*

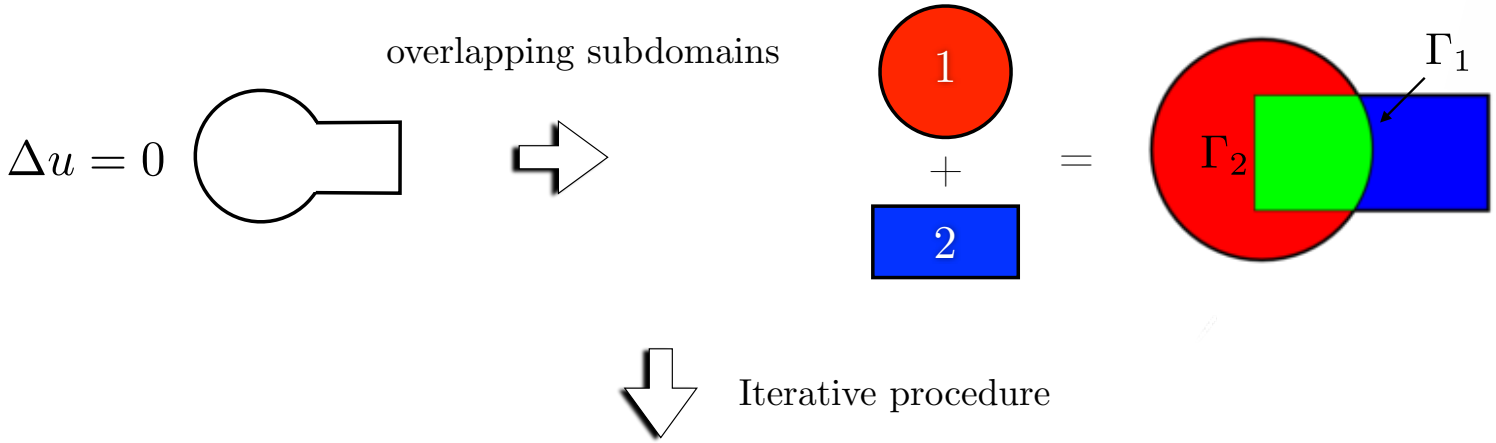
Die unter dem Namen Dirichletsches Princip bekannte Schlussweise, welche in gewissem Sinne als das Fundament des von Riemann entwickelten Zweiges der Theorie der analytischen Functionen angesehen werden muss, unterliegt, wie jetzt wohl allgemein zugestanden wird, hinsichtlich der Strenge sehr begründeten Einwendungen, deren vollständige Entfernung meines Wissens den Anstrengungen der Mathematiker bisher nicht gelungen ist. [1]





# Multiplicative Schwarz

## *Multiplicative Schwarz*



# Multiplicative *vs* Additive Schwarz

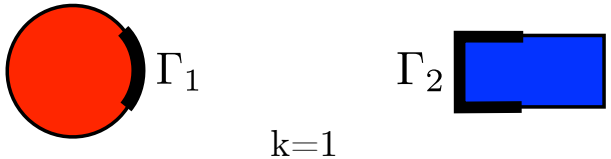
Multiplicative Schwarz is sequential

Additive Schwarz is parallel

# Additive Schwarz

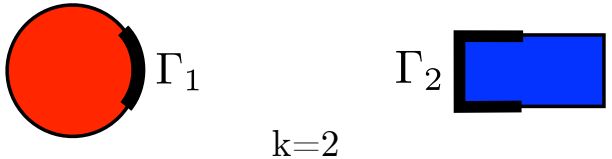
*Additive Schwarz*

$$\Delta u_1 = 0 \quad \text{in } \Omega_1 \qquad \Delta u_2 = 0 \quad \text{in } \Omega_2$$
$$u_1 = u_2 \quad \text{on } \Gamma_1 \qquad u_2 = u_1 \quad \text{on } \Gamma_2$$




k=1

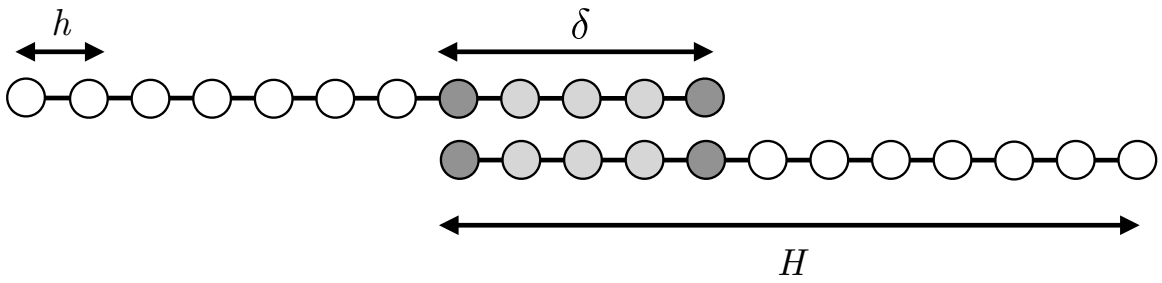
$$\Delta u_1 = 0 \quad \text{in } \Omega_1 \qquad \Delta u_2 = 0 \quad \text{in } \Omega_2$$
$$u_1 = u_2 \quad \text{on } \Gamma_1 \qquad u_2 = u_1 \quad \text{on } \Gamma_2$$




k=2

...

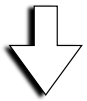
# Additive Schwarz




## *Good thing*

With coarse solver

$$\kappa(M_{AS}A) \leq C\left(1+\frac{H}{\delta}\right)$$



Condition number independent of mesh size $h$ and number of subdomains if $\frac{H}{\delta}$ is constant

## *Bad things*

- Work is duplicated in overlapping zone
- Cumbersome in parallel
- High cost

# Restricted Additive Schwarz (RAS)

***Restricted Additive Schwarz***

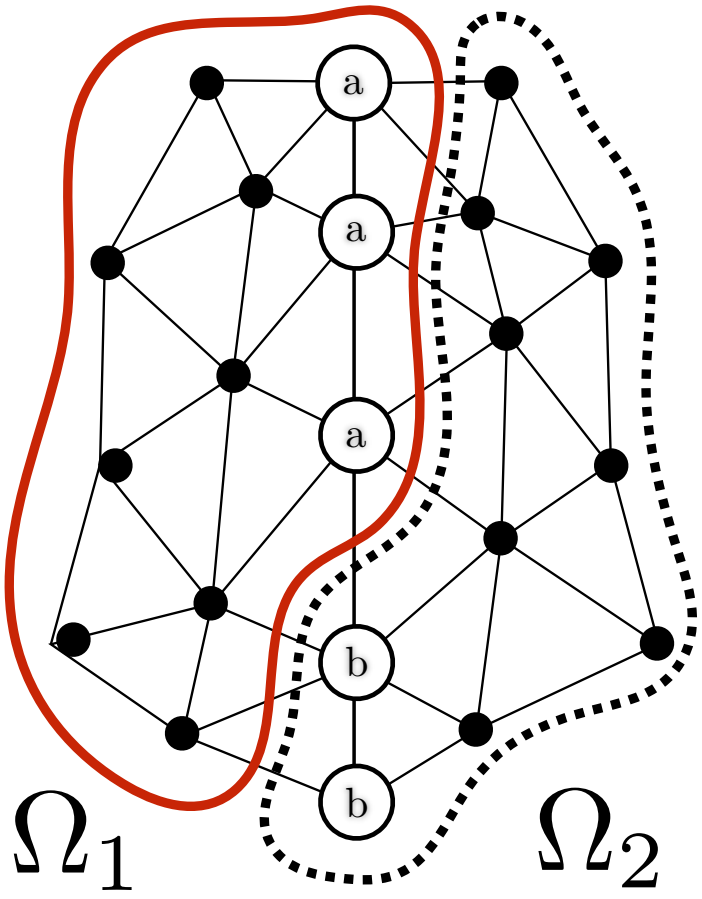
=

Schwarz with minimum overlap

and

unique solution in the overlapping zone

# Restricted Additive Schwarz (RAS)




Interior nodes

Shared interface nodes

a Own interface nodes of $\Omega_1$

b Own interface nodes of $\Omega_2$

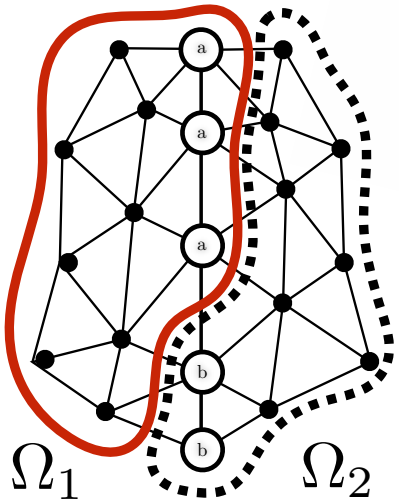
Nodes responsible for RAS in $\Omega_1$

Nodes responsible for RAS in $\Omega_2$

# Restricted Additive Schwarz (RAS)

Numbering:

$$x = (x_1 \;\; x_a \;\; x_b \;\; x_2)^t$$



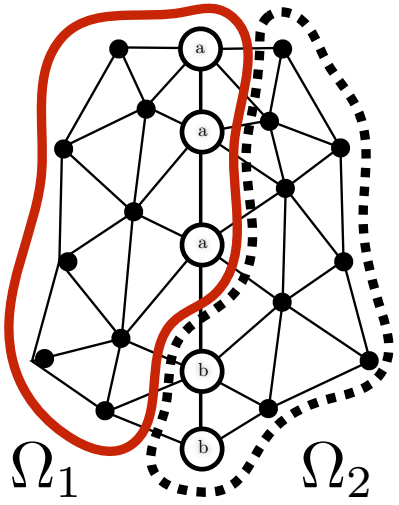

Global system:

$$\begin{pmatrix} A_{11} & A_{1a} & A_{1b} & 0 \\ A_{a1} & A_{aa} & A_{ab} & A_{a2} \\ A_{b1} & A_{ba} & A_{bb} & A_{b2} \\ 0 & A_{2a} & A_{2b} & A_{22} \end{pmatrix} \begin{pmatrix} x_1 \\ x_a \\ x_b \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_a \\ b_b \\ b_2 \end{pmatrix}$$

# Restricted Additive Schwarz (RAS)

$$\tilde{A}_1 = \begin{pmatrix} A_{11} & A_{1a} & A_{1b} \\ A_{a1} & A_{aa} & A_{ab} \\ A_{b1} & A_{ba} & A_{bb} \end{pmatrix}$$

Communication required!
A priori only on interface




$$\tilde{A}_2 = \begin{pmatrix} A_{aa} & A_{ab} & A_{a2} \\ A_{ba} & A_{bb} & A_{b2} \\ A_{2a} & A_{2b} & A_{22} \end{pmatrix}$$

# Restricted Additive Schwarz (RAS)

RAS: Solve the following preconditioned Richardson iteration

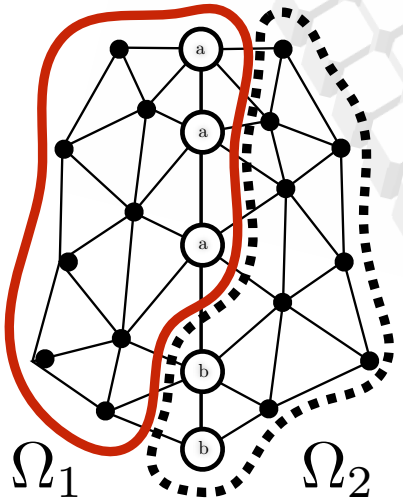
$$x^{p+1} = x^p + M(b - Ax)$$

$$\begin{pmatrix} x_1 \\ x_a \\ x_b \\ \hline x_2 \end{pmatrix}^{p+1}_1 = \begin{pmatrix} x_1 \\ x_a \\ x_b \\ \hline x_2 \end{pmatrix}^{p} + \left(\begin{array}{c|c} \tilde{A}_1^{-1} & 0 \\ \hline 0 & 0 \end{array}\right) \left[ \begin{pmatrix} b_1 \\ b_a \\ b_b \\ \hline b_2 \end{pmatrix} - \left(\begin{array}{c|c} \tilde{A}_1 & \begin{matrix} 0 \\ A_{a2} \\ A_{b2} \end{matrix} \\ \hline \begin{matrix} 0 A_{2a} A_{2b} \end{matrix} & A_{22} \end{array}\right) \begin{pmatrix} x_1 \\ x_a \\ x_b \\ \hline x_2 \end{pmatrix}^{p} \right]$$

$$\begin{pmatrix} x_1 \\ \hline x_a \\ x_b \\ x_2 \end{pmatrix}^{p+1}_2 = \begin{pmatrix} x_1 \\ \hline x_a \\ x_b \\ x_2 \end{pmatrix}^{p} + \left(\begin{array}{c|c} 0 & 0 \\ \hline 0 & \tilde{A}_2^{-1} \end{array}\right) \left[ \begin{pmatrix} b_1 \\ \hline b_a \\ b_b \\ b_2 \end{pmatrix} - \left(\begin{array}{c|c} A_{11} & A_{1a} A_{1b} 0 \\ \hline \begin{matrix} A_{a1} \\ A_{b1} \\ 0 \end{matrix} & \tilde{A}_2 \end{array}\right) \begin{pmatrix} x_1 \\ \hline x_a \\ x_b \\ x_2 \end{pmatrix}^{p} \right]$$

Precond. $M$ RHS $b$ Matrix $A$

Residual



# Restricted Additive Schwarz (RAS)

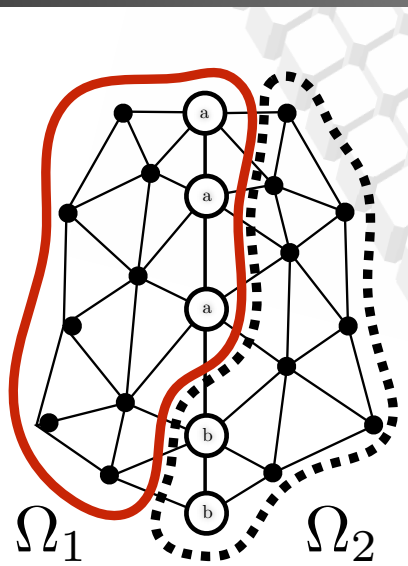
Last system is equivalent to solving:

- Solve **exactly** in $\Omega_1$ using solution in $\Omega_2$ as **Dirichlet** condition

$$\begin{pmatrix} x_1 \\ x_a \\ x_b \\ \hline x_2 \end{pmatrix}_1^{p+1} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ \hline x_2^p \end{pmatrix} + \begin{pmatrix} \tilde{A}_1^{-1} \left[ \begin{pmatrix} b_1 \\ b_a \\ b_b \end{pmatrix} - \begin{pmatrix} 0 \\ A_{a2}x_2^p \\ A_{b2}x_2^p \end{pmatrix} \right] \\ \hline 0 \end{pmatrix}$$

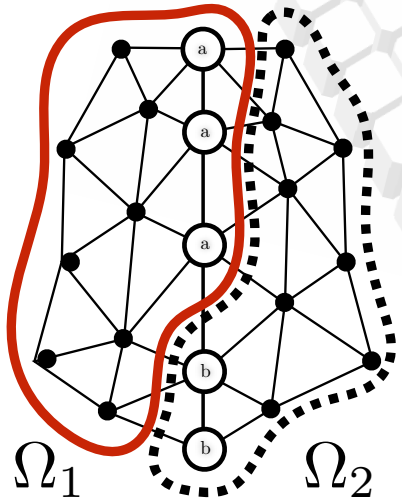
- Solve **exactly** in $\Omega_2$ using solution in $\Omega_1$ as **Dirichlet** condition

$$\begin{pmatrix} x_1 \\ \hline x_a \\ x_b \\ x_2 \end{pmatrix}_2^{p+1} = \begin{pmatrix} x_1^p \\ \hline 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ \hline \tilde{A}_2^{-1} \left[ \begin{pmatrix} b_a \\ b_b \\ b_2 \end{pmatrix} - \begin{pmatrix} A_{a1}x_1^p \\ A_{b1}x_1^p \\ 0 \end{pmatrix} \right] \end{pmatrix}$$



# Restricted Additive Schwarz (RAS)

To have a unique update on the interface, subdomain 1 is responsible for $a$ and subdomain 2 is responsible for $b$




$$\begin{pmatrix} x_1 \\ x_a \\ \hline x_b \\ x_2 \end{pmatrix}_1^{p+1} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ \hline x_2^p \end{pmatrix} + \begin{pmatrix} \tilde{A}_1^{-1} \left[ \begin{pmatrix} b_1 \\ b_a \\ b_b \end{pmatrix} - \begin{pmatrix} 0 \\ A_{a2}x_2^p \\ A_{b2}x_2^p \end{pmatrix} \right] \\ \hline 0 \end{pmatrix}$$

$$\begin{pmatrix} x_1 \\ \hline x_a \\ x_b \\ x_2 \end{pmatrix}_2^{p+1} = \begin{pmatrix} x_1^p \\ \hline 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ \hline \tilde{A}_2^{-1} \left[ \begin{pmatrix} b_a \\ b_b \\ b_2 \end{pmatrix} - \begin{pmatrix} A_{a1}x_1^p \\ A_{b1}x_1^p \\ 0 \end{pmatrix} \right] \end{pmatrix}$$

# Restricted Additive Schwarz (RAS)

RAS Preconditioning. Solve $M\ z = r$

- Compute local matrices $\tilde{A}_1 = \begin{pmatrix} A_{11} & A_{1a} & A_{1b} \\ A_{a1} & A_{aa}^{(1)} & A_{ab}^{(1)} \\ A_{b1} & A_{ba}^{(1)} & A_{bb}^{(1)} \end{pmatrix}$ $\tilde{A}_2 = \begin{pmatrix} A_{aa}^{(2)} & A_{ab}^{(2)} & A_{a2} \\ A_{ba}^{(2)} & A_{bb}^{(2)} & A_{b2} \\ A_{2a} & A_{2b} & A_{22} \end{pmatrix}$
- Exchange boundary values $A_{aa}^{(i)}$, $A_{ab}^{(i)}$, $A_{ba}^{(i)}$, $A_{bb}^{(i)}$
- Assemble local matrices $\tilde{A}_1 = \begin{pmatrix} A_{11} & A_{1a} & A_{1b} \\ A_{a1} & A_{aa} & A_{ab} \\ A_{b1} & A_{ba} & A_{bb} \end{pmatrix}$ $\tilde{A}_2 = \begin{pmatrix} A_{aa} & A_{ab} & A_{a2} \\ A_{ba} & A_{bb} & A_{b2} \\ A_{2a} & A_{2b} & A_{22} \end{pmatrix}$
- Compute inverse matrices $\tilde{A}_i^{-1}$
- Solve $z_i = \tilde{A}_i^{-1} r$
- Exchange solution $z_i$ to impose $z = z_{i,\text{own}}$

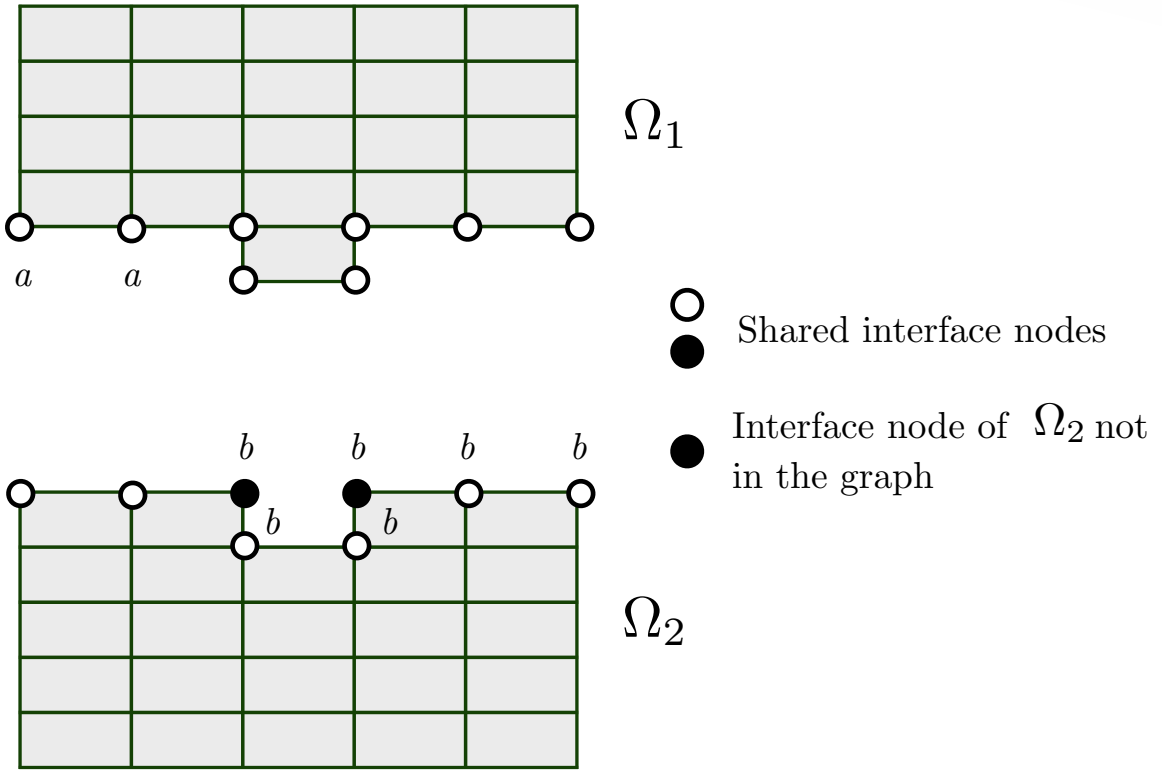
# BUT...!

But interface graph may be incomplete!

$$\tilde{A}_1 = \begin{pmatrix} A_{11} & A_{1a} & A_{1b} \\ A_{a1} & A_{aa} & A_{ab} \\ A_{b1} & A_{ba} & A_{bb} \end{pmatrix}$$

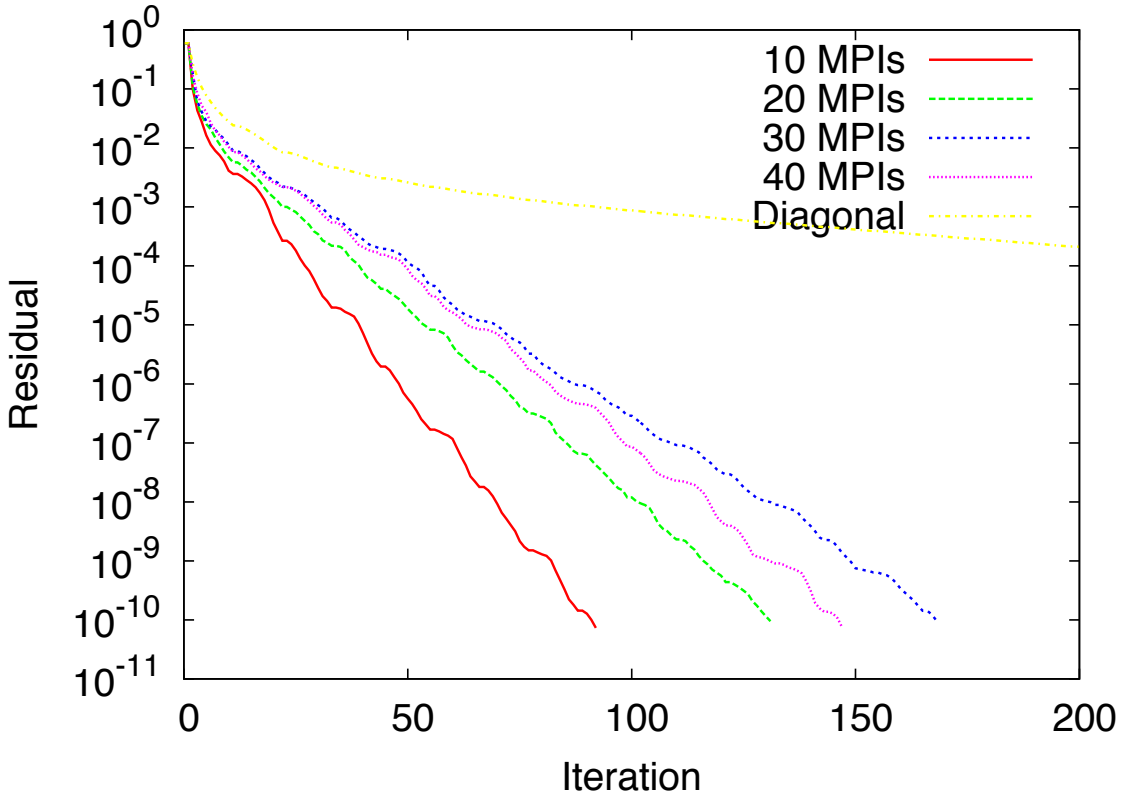
$\Omega_2$ cannot fill in $A_{bb}$ completely

$$\tilde{A}_2 = \begin{pmatrix} A_{aa} & A_{ab} & A_{a2} \\ A_{ba} & A_{bb} & A_{b2} \\ A_{2a} & A_{2b} & A_{22} \end{pmatrix}$$

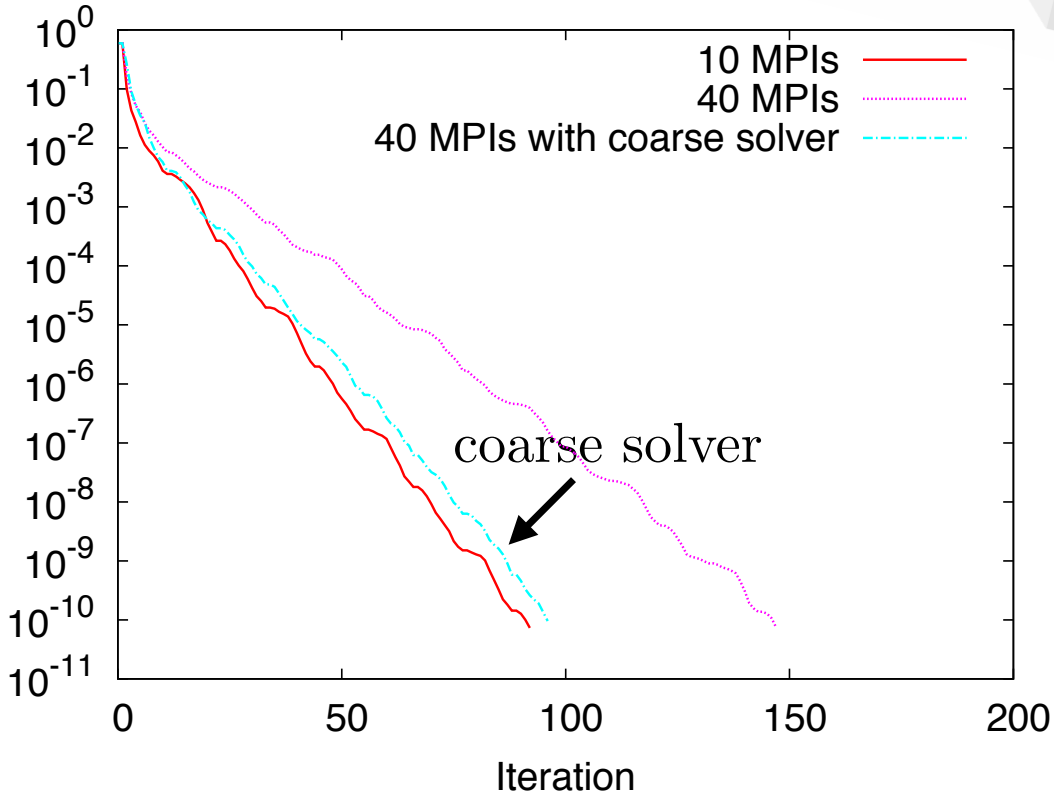


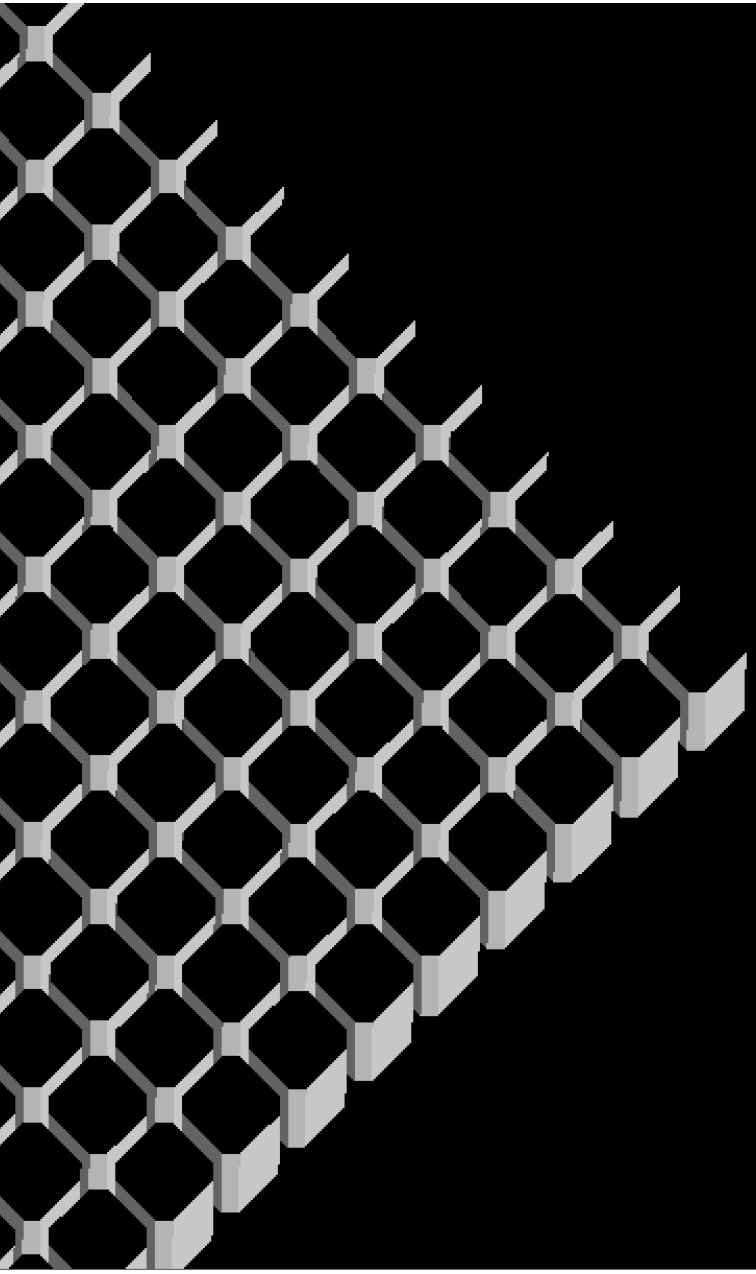
# Example

Without coarse solver, # iterations increases with number of subdomains

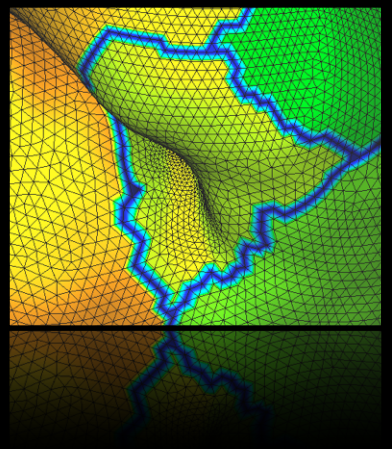


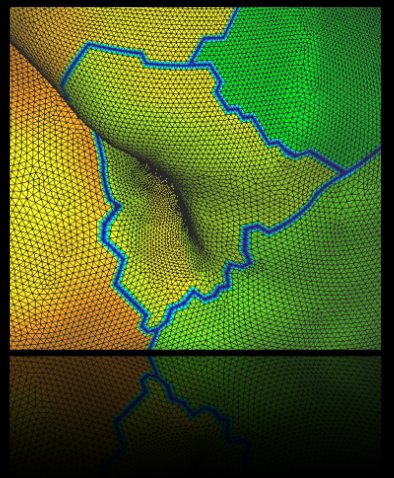

With coarse solver



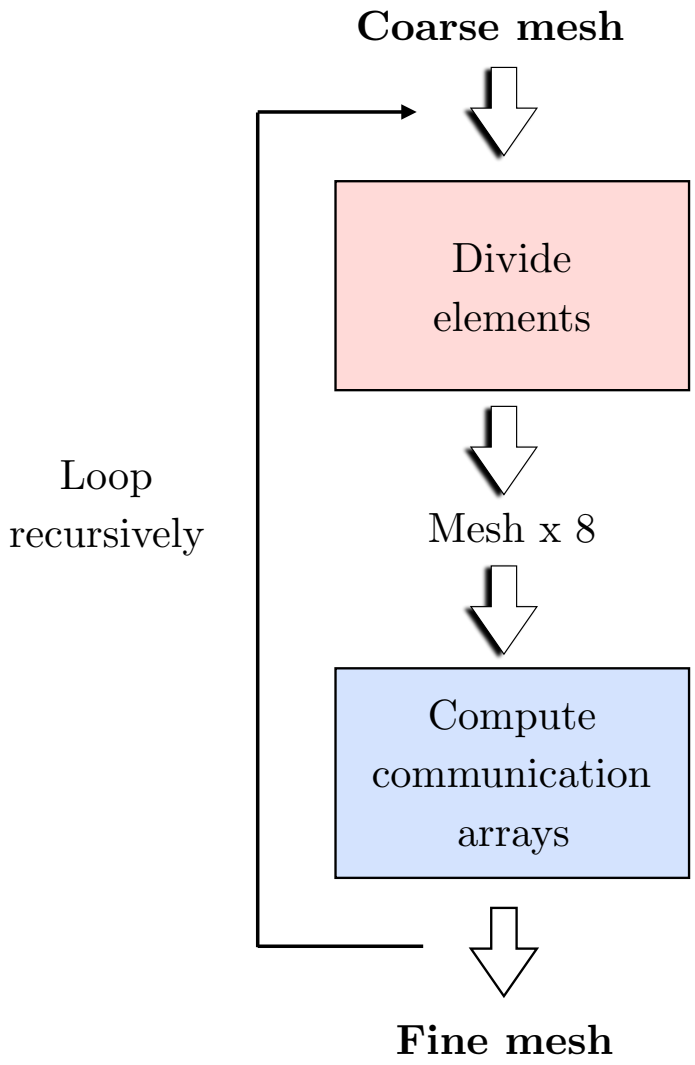


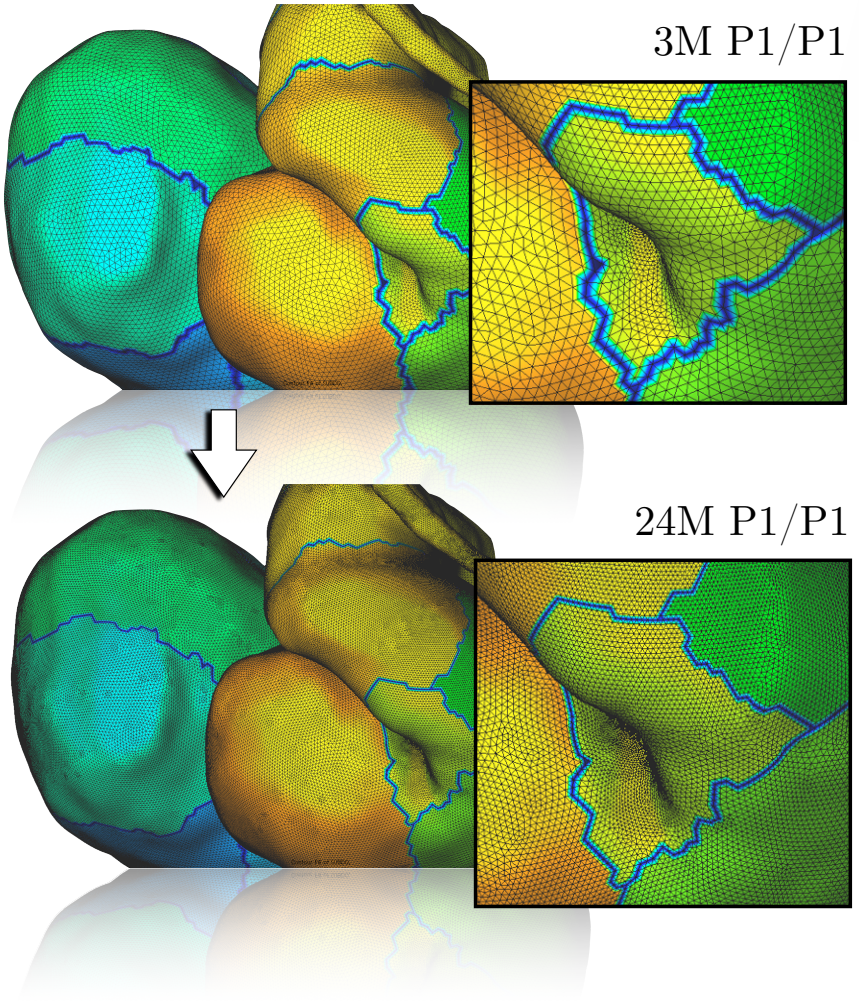
# Mesh Multiplication



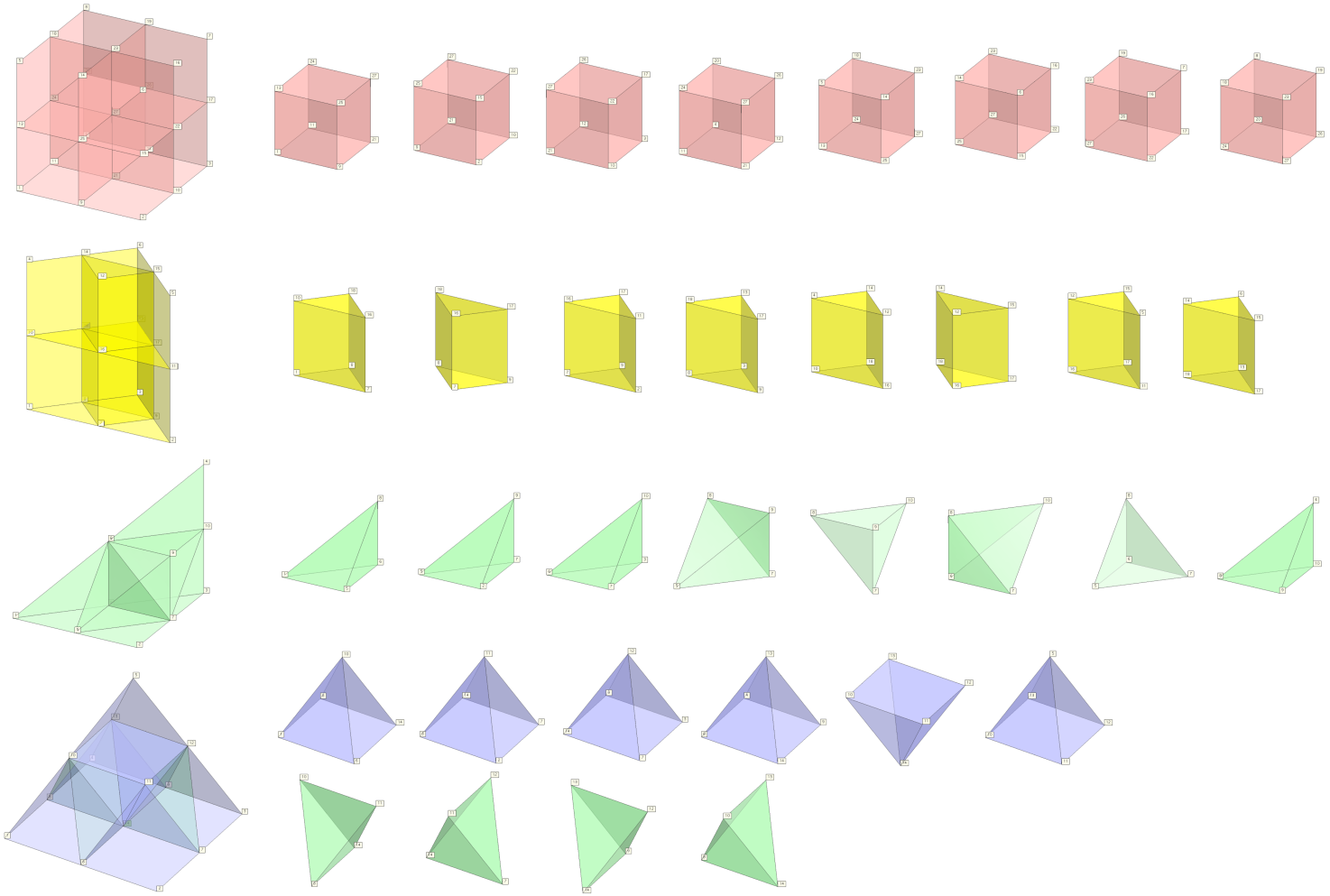


# An example: Mesh Multiplication



# An example: Mesh Multiplication



# An example: Mesh Multiplication

**A. Obtained previously**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 344 | 234 | 4 | 11 | 4 | 5 | 10 |
| 2 | 2 | 543 | 121 | 1 | 10 | | | |
| 3 | 5 | 124 | 435 | 4 | 3 | 4 | 2 | 11 |
| 4 | 6 | 432 | 12 | 2 | 4 | 5 | | |
| 5 | 7 | 134 | 56 | 2 | 2 | 3 | | |
| 6 | 9 | 65 | 23 | 3 | 11 | 10 | 3 | |

**LEDGB**: boundary edge table

**B. Reorder**

Order boundary edge nodes wrt global numbering
Change also edge table

original global

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 234 | 344 | 4 | 11 | 4 | 5 | 10 |
| 2 | 2 | 121 | 534 | 1 | 10 | | | |
| 3 | 5 | 124 | 435 | 4 | 3 | 4 | 2 | 11 |
| 4 | 6 | 12 | 432 | 2 | 4 | 5 | | |
| 5 | 7 | 56 | 123 | 2 | 2 | 3 | | |
| 6 | 9 | 23 | 65 | 3 | 11 | 10 | 3 | |

**LEDGB**: boundary edge table

local edge nodes local new node

| | | | | |
|---|---|---|---|---|
| 1 | 11 | 12 | 85 | 1 |
| 2 | 34 | 26 | 86 | 2 |
| 3 | 11 | 24 | 87 | 0 |
| 4 | ... | ... | -88 | 0 |
| 5 | ... | ... | 89 | 3 |
| 6 | 23 | 14 | 90 | 4 |
| 7 | ... | ... | 91 | 5 |
| 8 | | | -92 | 0 |
| 9 | | | 93 | 6 |
| 10 | | | -94 | 0 |
| 11 | | | 95 | 0 |

Example:
Boundary edge in common with subd. 4

**LEDGG**: edge table

**C. Fill in edge boundary with subdomain INEIG**

| | |
|---|---|
| 1 | 85 |
| 2 | 89 |
| 3 | 90 |

**LEDGB_LOC**: boundary edge table

**D. Order edges using global and unique numbering**

| | |
|---|---|
| 1 | 89 |
| 2 | 85 |
| 3 | 90 |

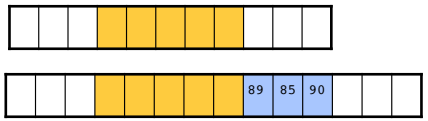
| | |
|---|---|
| 344 | 234 |
| 124 | 435 |
| 432 | 12 |

| | |
|---|---|
| 124 | 435 |
| 344 | 234 |
| 432 | 12 |

**LEDGB_LOC**: boundary edge table

**E. Modify communication array**

E1. Copy old permutation array and reallocate it
E2. Compute new permutation array and bounds




**F. Complete renumbering**

Renumber the mesh nodes and all the nodal arrays

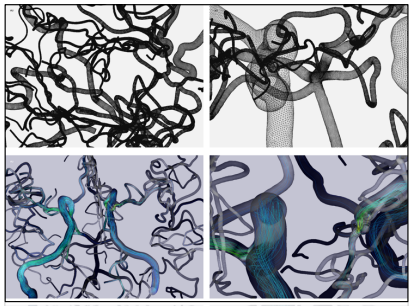
**G. Define a global and unique numbering**

Permutation array between my local numbering obtained in F and a global and unique numbering.
This is useful for:
- Perform MM recursively
- Postprocess purpose

# An example: Mesh Multiplication

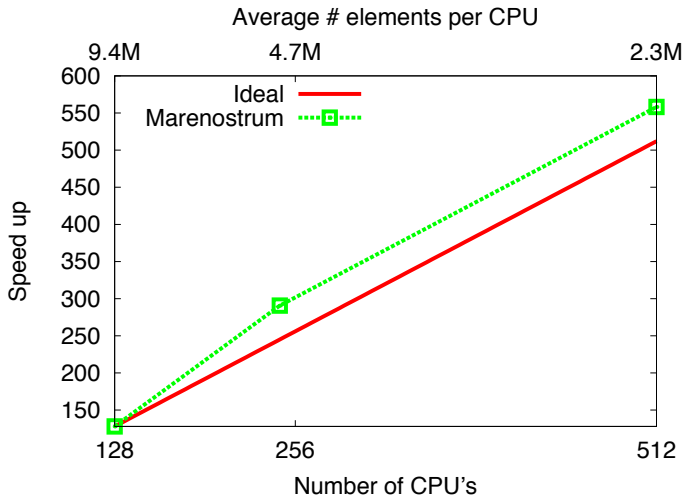
## Marenostrum (10240 PowerPC970)

19M P1/P1
⇩
152M P1/P1
⇩
**1.2B P1/P1**
⇩
9.7B P1/P1
⇩
Not enough memory!!!
77.8B P1/P1



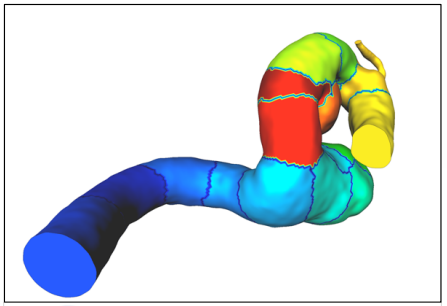
| #CPU's | Time MM | Speedup |
|---|---|---|
| 128 | 109s | 1 |
| 256 | 48s | 2.3 |
| 512 | 25s | 4.4 |

19M to 1.2B with MM (2)



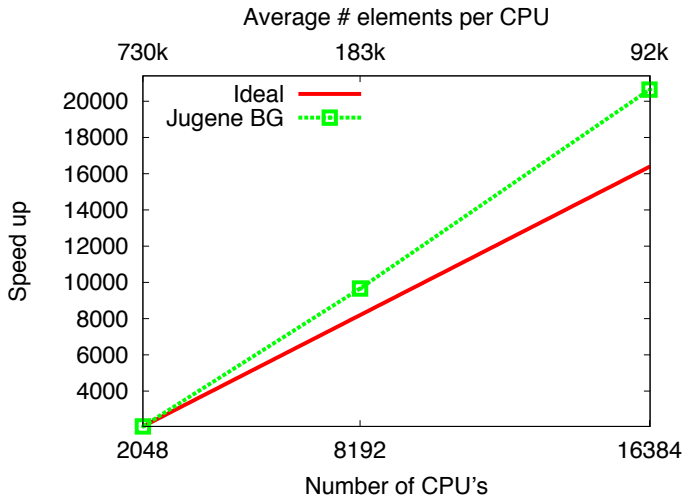

## Jugene BlueGene (294912 PowerPC450)

2.9M P1/P1
⇩
23.5M P1/P1
⇩
188M P1/P1
⇩
**1.5B P1/P1**



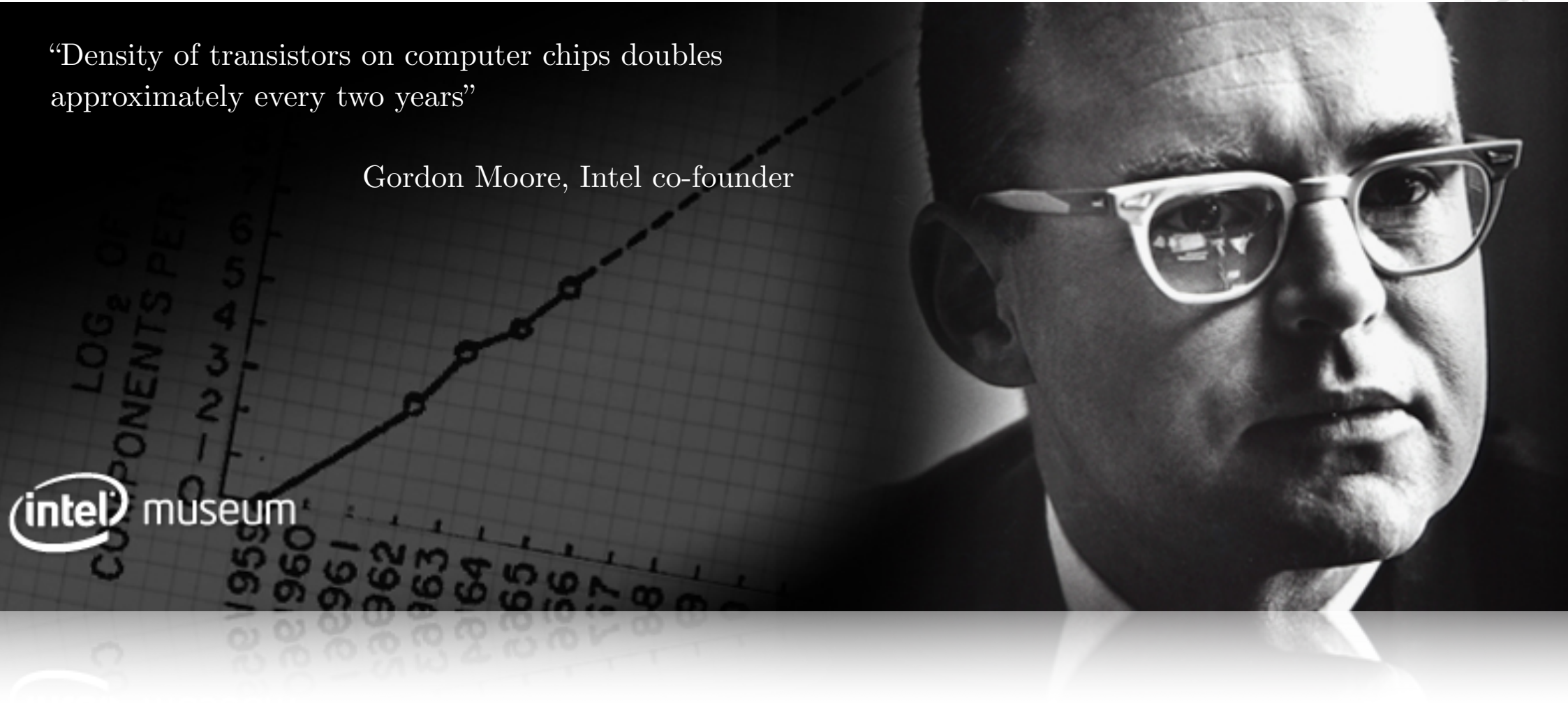
| #CPU's | Time MM | Speedup |
|---|---|---|
| 2048 | 14.0s | 1 |
| 8192 | 3.0s | 4.7 |
| 16384 | 1.4s | 10.1 |

3M to 1.5B with MM (3)

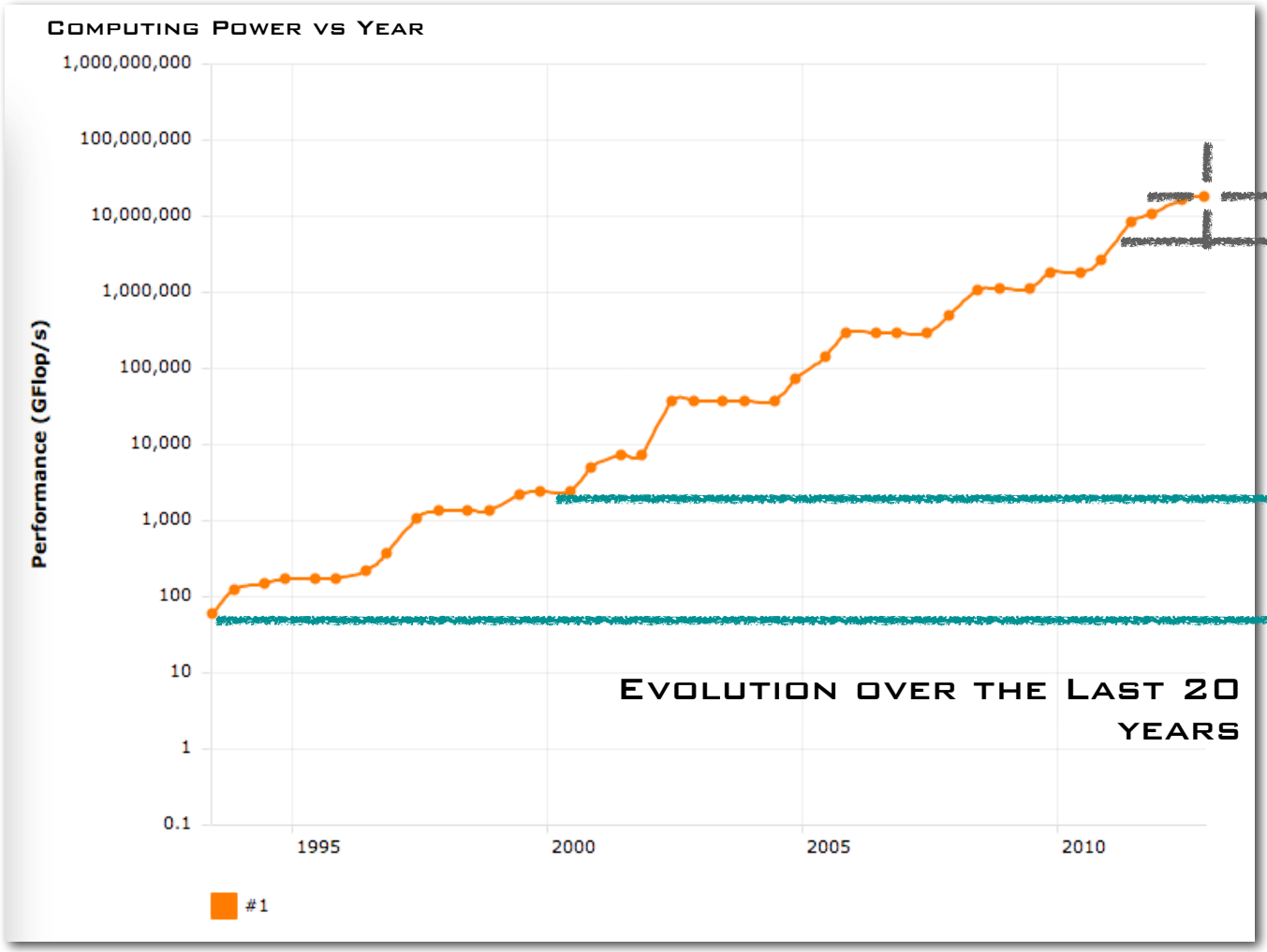



# Hardware

# About the future: Hardware



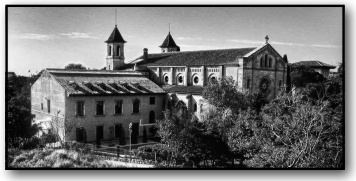
# Where HPC is now




top500.org
The 500 largest supercomputers

Titan

Marenostrum



nVidia K20X

iPad 4

# About the future: Hardware

*HIGH POWER CONSUMPTION*

17808 kW

Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P

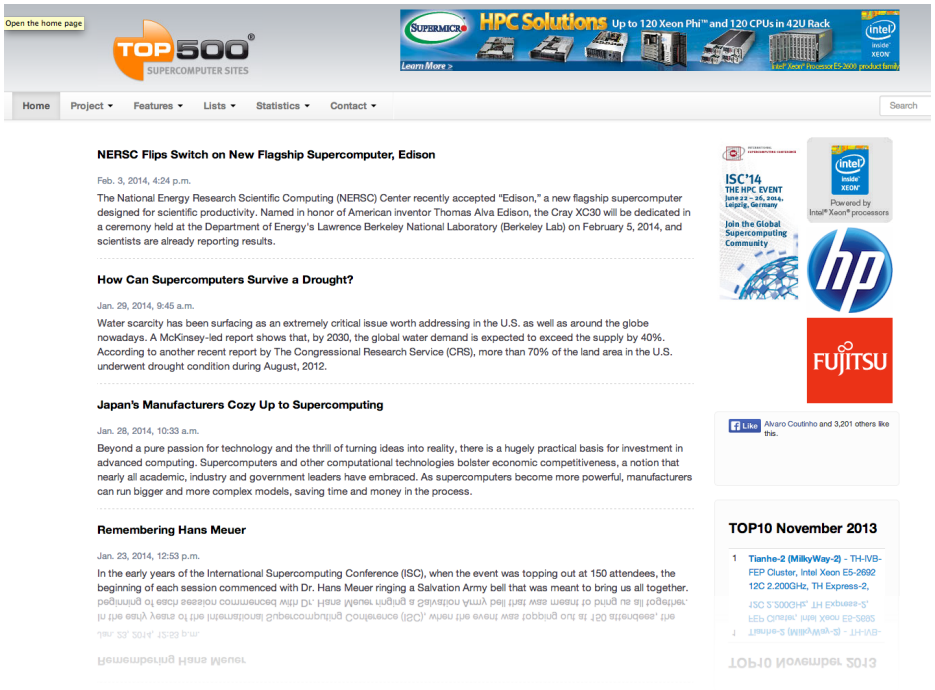
Toledo, Spain
Population: 80,000

# About the future: Hardware

TOP500 *vs* GREEN500

GREEN500: A better way of measuring performance

The purpose of the Green500 is to provide a ranking of the most energy-efficient supercomputers in the world




# About the future: Hardware

*SOME DARK THOUGHTS*

"They expect 1000 times more computations per second within a decade. If we were to try to accomplish this with today's technology, we would eat up the world's total electric power within five years. Total electric power!"

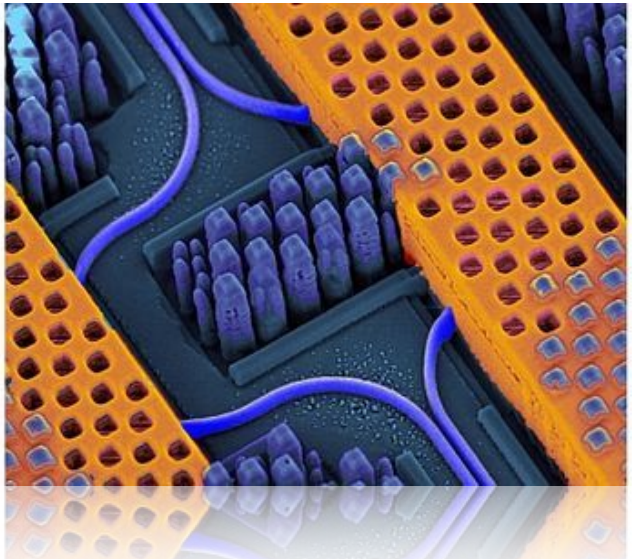
Bernd Hoefflinger (editor of Chips 2020 book)

# About the future: Hardware

*MORE DARK THOUGHTS*

"The usable limit for semiconductor process technology will be reached when chip process geometries shrink to be smaller than 20 nanometers (nm), to 18nm nodes. At those nodes, the industry will start getting to the point where semiconductor manufacturing tools are too expensive to depreciate with volume production, i.e., their costs will be so high, that the value of their lifetime productivity can never justify it."

Len Jelinek, director and chief analyst for semiconductor manufacturing iSuppli

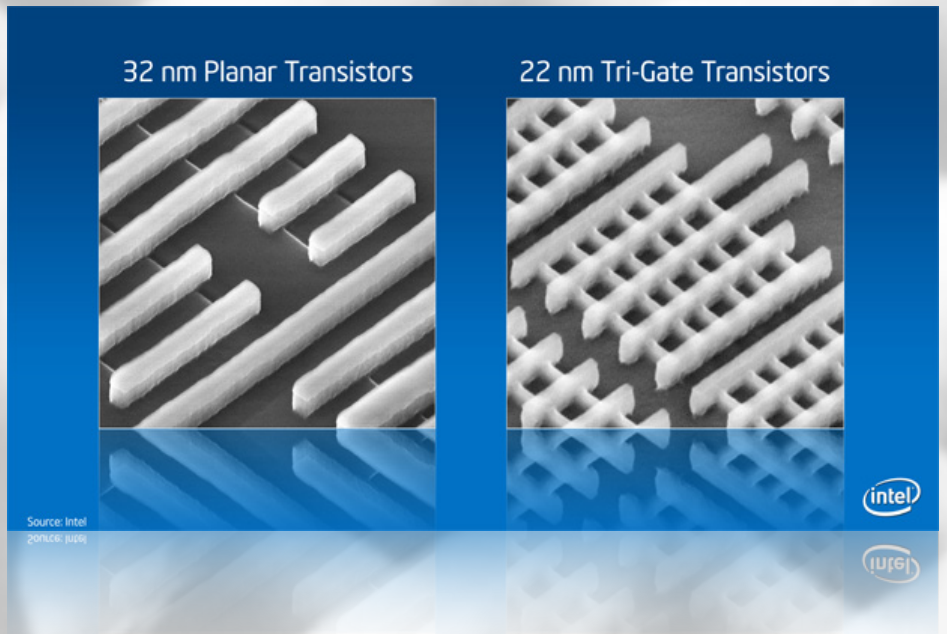


# About the future: Hardware

*...BUT THERE IS ALWAYS LIGHT AT THE END OF THE TUNNEL*

"There's always physical limits to everything, but you can always come up with clever ways... for example, there's nothing that says I can't take two dies and stack them on top of each other so I can grow Moore's Law in the third dimension"

Steve Pawlowski, senior fellow and head honcho on exascale research

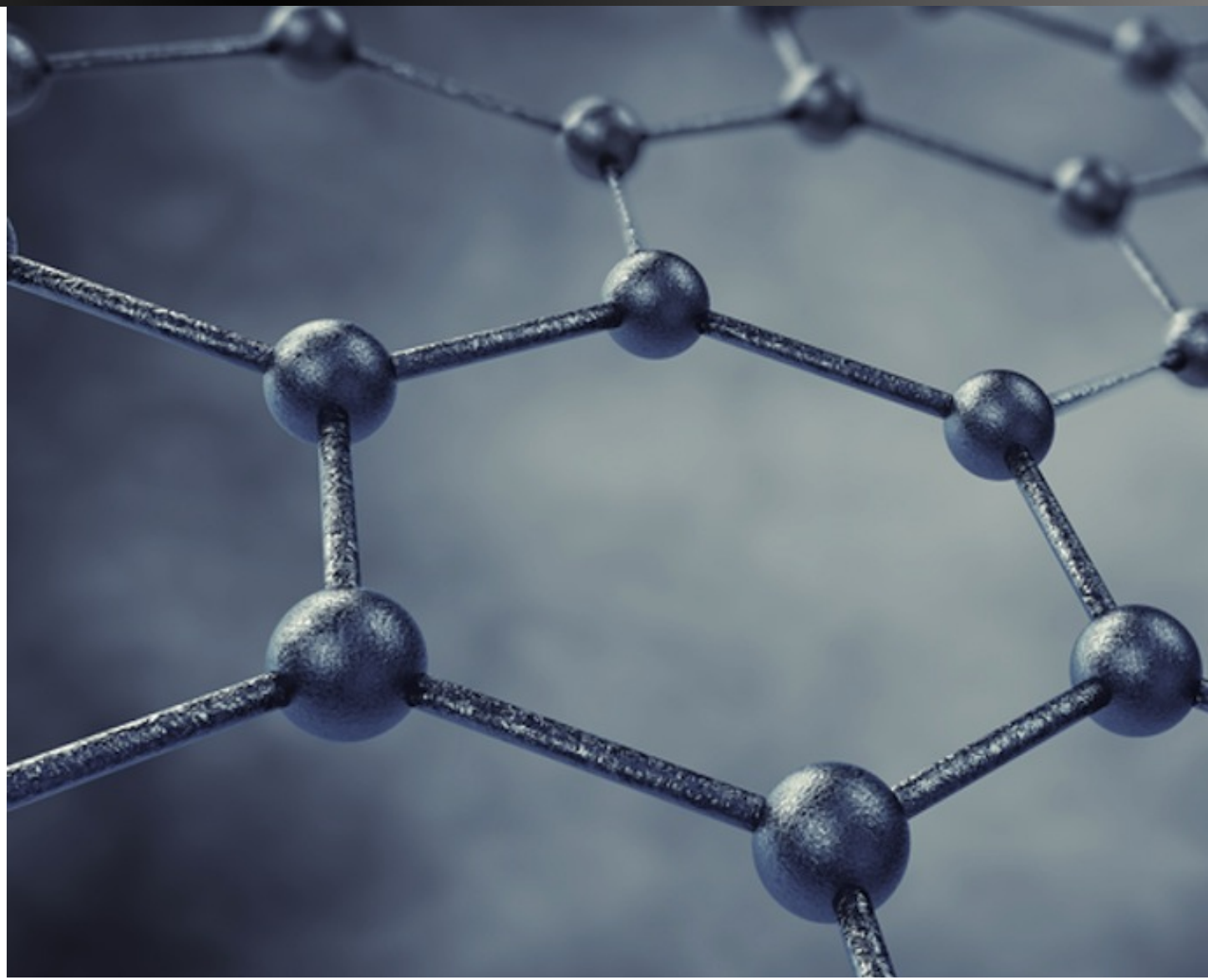


# About the future: Hardware

*NEW WAYS & TECHNOLOGIES*

***GRAPHENE***

Integrated circuits
Electrochromic devices
Transparent conducting electrodes
Desalination
Solar cells
Single-molecule gas detection
Graphene nanoribbons
Graphene quantum dots
Graphene transistors
Graphene optical modulators
Thermal management materials
Ultracapacitors
Engineered piezoelectricity
Graphene biodevices
...



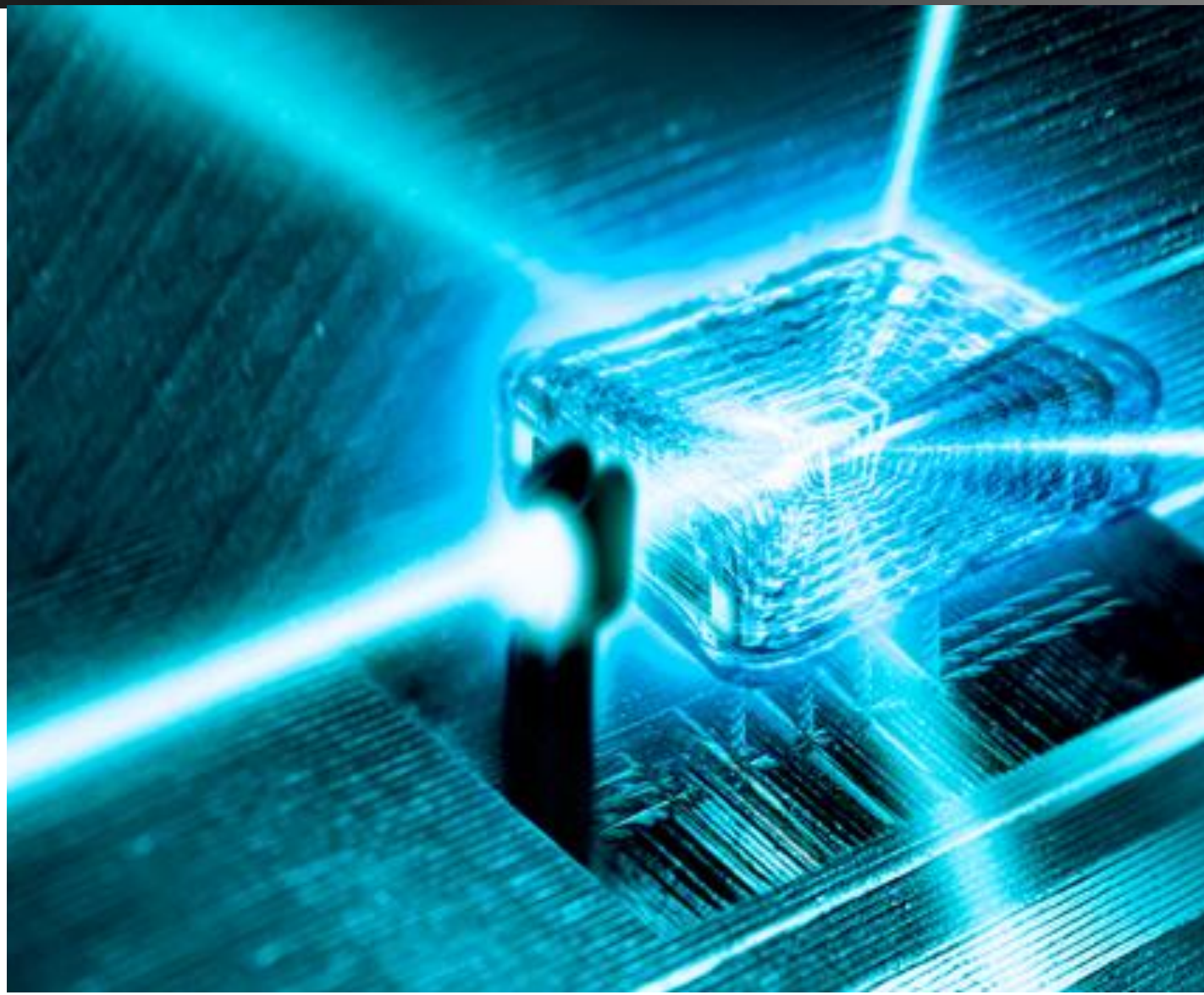
# About the future: Hardware

*NEW WAYS & TECHNOLOGIES*

***QUANTUM COMPUTING***

Several orders of magnitude
faster than current technology

Linear equations system solving
Criptography
Web Search
...




Software

# Beyong classical solvers

Classical solvers

- Lots of exhange
- Synchronization points
- Strong scalability has been good
- Poor weak scalability

Classical Jacobi

$$\mathbf{x}^k = \mathbf{x}^{k-1} + \mathbf{D}^{-1}(\mathbf{b} - \mathbf{A}\mathbf{x}^{k-1})$$

$$x_i^k = \frac{1}{A_{ii}} \left( b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} A_{ij} x_j^{k-1} \right)$$

Synchronization

Desynchronize

Chaotic Jacobi

$$x_i^k = \frac{1}{A_{ii}} \left( b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} A_{ij} x_j^{k-s} \right)$$

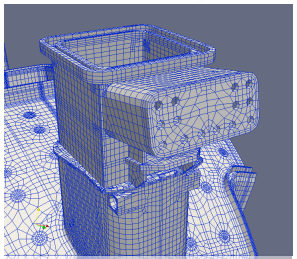
# About the future: Software

*CHALLENGES*

Multiphysics: code coupling, multiscale

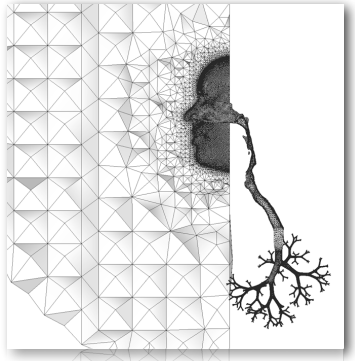
Alya ⇔ CODE SATURNE

Meshing: billions of elements



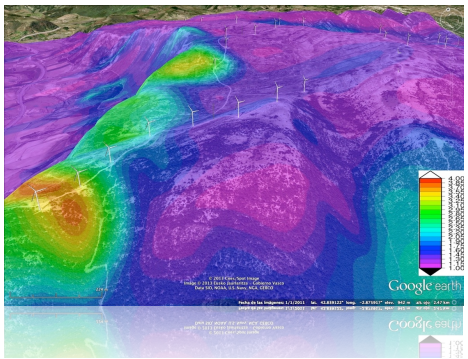
On the fly meshing?

Verly large scale simulations



22B elements
100,000 CPUs

Visualization: in situ, large data sets



*PREGUNTAS?*

Frédéric Magoulès
François-Xavier Roux
Guillaume Houzeaux

Cálculo Científico Paralelo

CIMNE

# Cálculo Científico Paralelo

Frédéric Magoulès
François-Xavier Roux
Guillaume Houzeaux