MATH 676

–

Finite element methods in scientific computing

Wolfgang Bangerth, Texas A&M University

Lecture 41.75:

Parallelization on a cluster of distributed memory machines

Part 4: Parallel solvers and preconditioners

General approach to parallel solvers

Historically, there are three general approaches to solving PDEs in parallel:

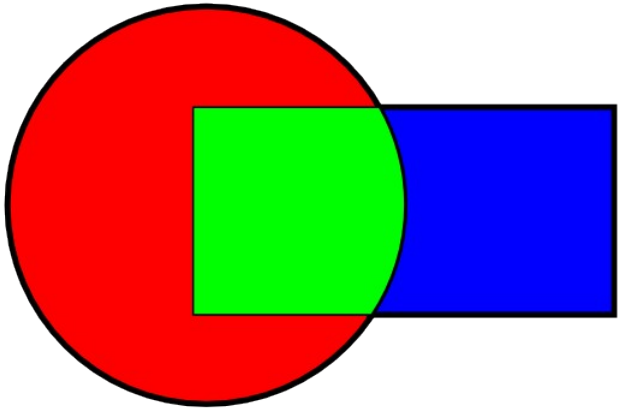
- *Domain decomposition:*
  - Split the domain on which the PDE is posed
  - Discretize and solve (small) problems on subdomains
  - Iterate out solutions

- *Global solvers:*
  - Discretize the global problem
  - Receive one (very large) linear system
  - Solve the linear system in parallel

- *A compromise: Mortar methods*

Domain decomposition

Historical idea: Consider solving a PDE on such a domain:

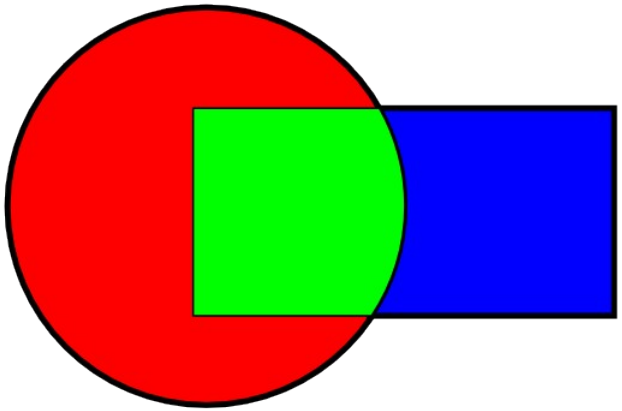


Source: Wikipedia

Note: We know how to solve PDEs analytically on each part of the domain.

Domain decomposition

Historical idea: Consider solving a PDE on such a domain:

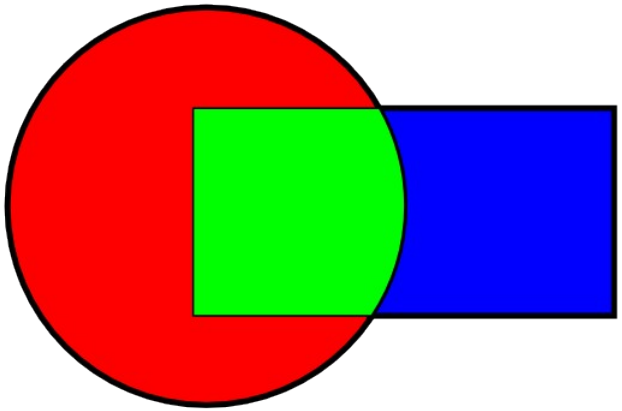


Approach (Hermann Schwarz, 1870):

- Solve on circle using arbitrary boundary values, get u^1
- Solve on rectangle using u^1 as boundary values, get u^2
- Solve on circle using u^2 as boundary values, get u^3
- Iterate (proof of convergence: Mikhlin, 1951)

Domain decomposition

Historical idea: Consider solving a PDE on such a domain:

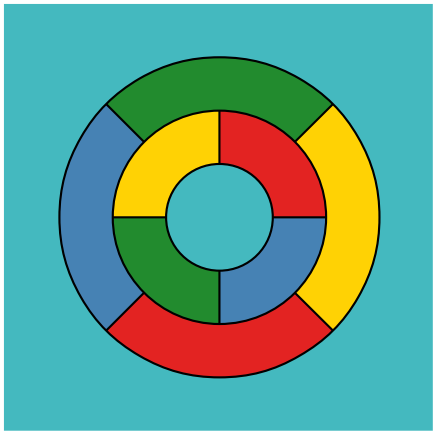


This is called the *Alternating Schwarz* method. When discretized:

- Shape of subdomains no longer important
- Easily generalized to many subdomains
- This is called *Overlapping Domain Decomposition* method

Domain decomposition

Alternative: Non-overlapping domain decomposition

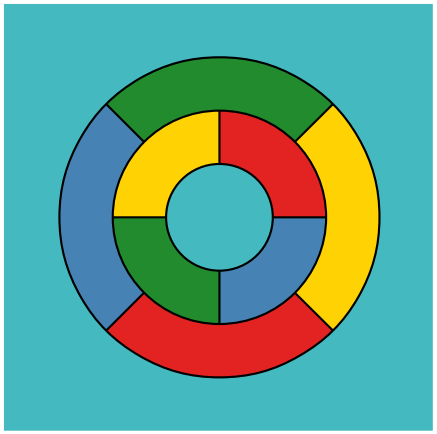


The following does not work:

- Solve on subdomain 1 using arbitrary b.v., get u^1
- Solve on subdomain 2 using u^1 as Dirichlet b.v., get u^2
- …
- Solve on subdomain 1 using u^N as Dirichlet b.v., get u^{N+1}
- Iterate

Domain decomposition

Alternative: Non-overlapping domain decomposition



This does work (Dirichlet-Neumann iteration):

- Solve on subdomain 1 using arbitrary b.v., get u^1
- Solve on subdomain 2 using u^1 as Dirichlet b.v., get u^2
- …
- Solve on subdomain 1 using u^N as *Neumann* b.v., get u^{N+1}
- Iterate

Domain decomposition

History's verdict:

- Some beautiful mathematics came of it
- Iteration converges too slowly
- Particularly with large numbers of subdomains (lack of global information exchange)
- Does not play nicely with modern ideas for discretization:
  – mesh adaptation
  – hp adaptivity

Global solvers

General approach:

- Mesh the entire domain in *one* mesh
- Partition the mesh between processors

- Each processor discretizes its part of the domain

- Obtain one very large linear system
- Solve it with an iterative solver
- Apply a preconditioner to the whole system

- Adapt mesh as necessary, rebalance between processors

Global solvers

General approach:

- Mesh the entire domain in *one* mesh
- Partition the mesh between processors

- Each processor discretizes its part of the domain

- Obtain one very large linear system
- Solve it with an iterative solver
- Apply a preconditioner to the whole system

- Adapt mesh as necessary, rebalance between processors

Note: Each step here requires communication; much more sophisticated software necessary!

Global solvers

Pros:

- Convergence independent of subdivision into subdomains (if good preconditioner)
- Load balancing with adaptivity not a problem
- Has been shown to scale to 100,000s of processors

Cons:

- Requires *much* more sophisticated software
- Relies on *iterative* linear solvers
- Requires sophisticated preconditioners

But: Powerful software libraries available for all steps.

Global solvers

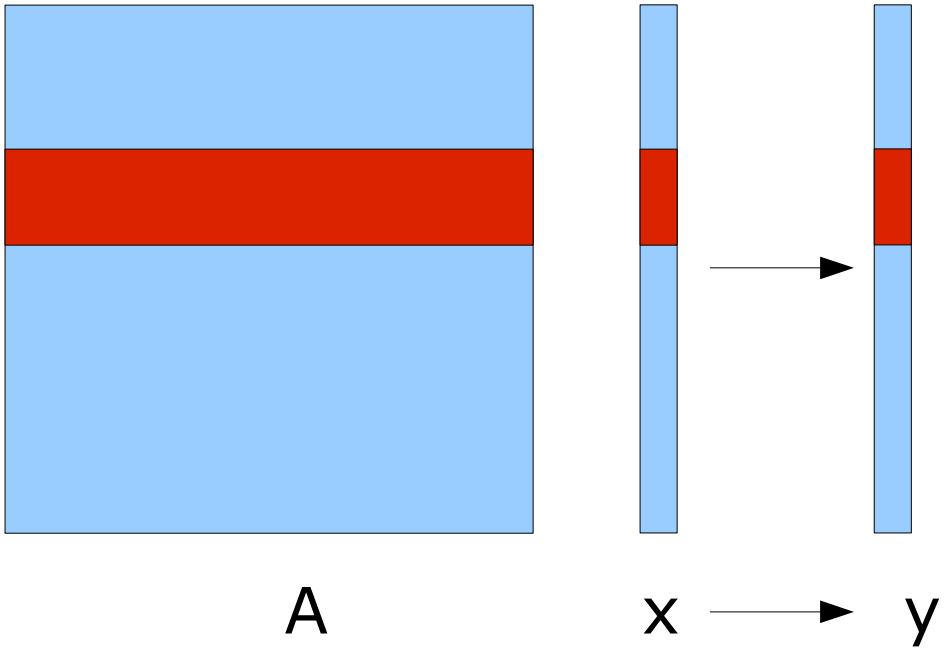
Examples for a few necessary steps:

- Matrix-vector products in iterative solvers (Point-to-point communication)
- Dot product synchronization
- Available preconditioners

Matrix-vector product

What does processor *P* need:

- Graphical representation of what *P* owns:

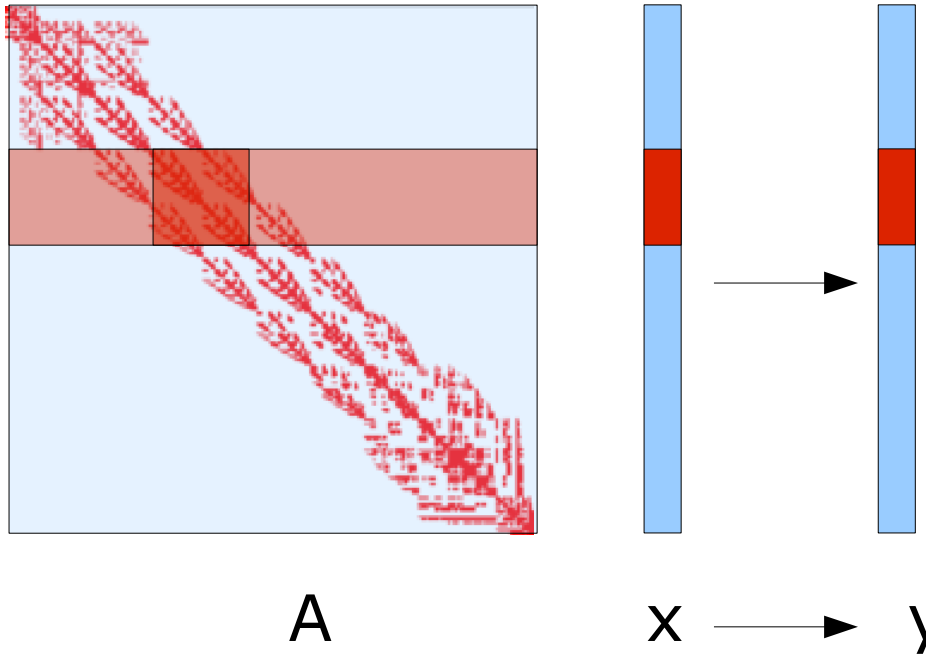


- To compute the *locally owned* elements of *y*, processor *P* needs **all** elements of *x*
- All processors need to send their share of *x* to everyone

Matrix-vector product

What does processor *P* need:

- **But:** Finite element matrices look like this:

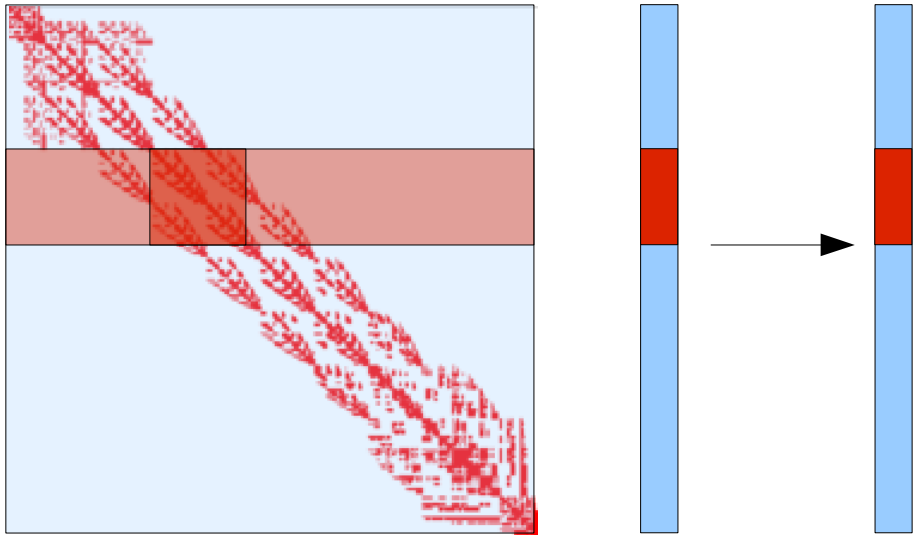


For the *locally owned* elements of y, processor P needs **all** x_j for which there is a nonzero A_{ij} for a locally owned row i.

Matrix-vector product

What does processor *P* need to compute its part of *y*:

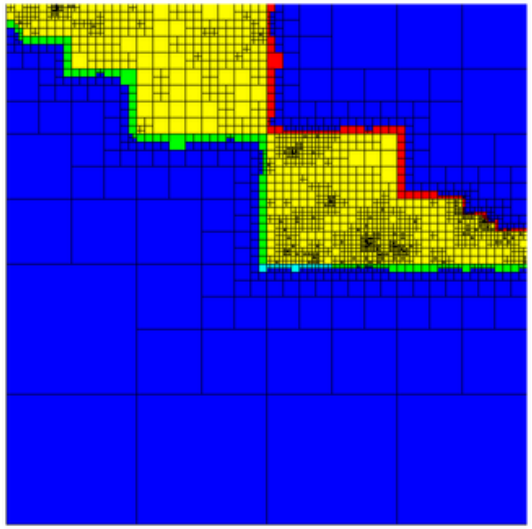
- All elements x_j for which there is a nonzero A_{ij} for a locally owned row i.
- In other words, if x_i is a locally owned DoF, we need all x_j that couple with x_i
- These are exactly the *locally relevant degrees of freedom*
- They live on *ghost cells*



Matrix-vector product

What does processor *P* need to compute its part of *y*:

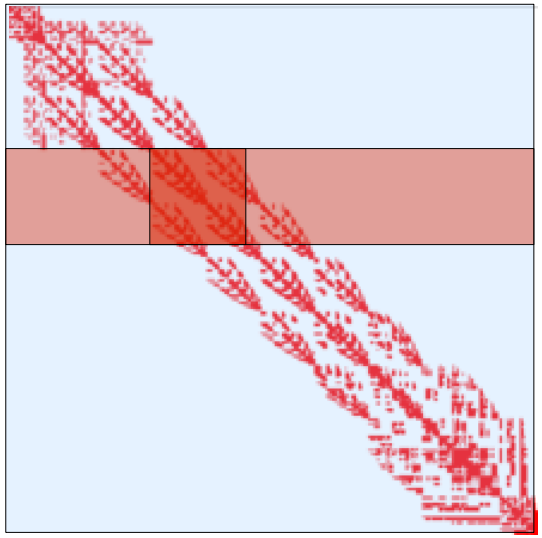
- All elements x_j for which there is a nonzero A_{ij} for a locally owned row i.
- In other words, if x_i is a locally owned DoF, we need all x_j that couple with x_i
- These are exactly the *locally relevant degrees of freedom*
- They live on *ghost cells*



Matrix-vector product

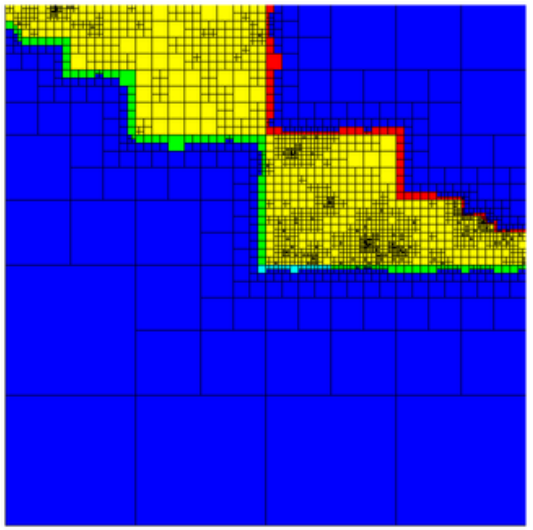
Parallel matrix-vector products for sparse matrices:

- Requires determining which elements we need from which processor
- Exchange this up front once



Performing matrix-vector product:

- Send vector elements to all processors that need to know
- Do local product (dark red region)
- Wait for data to come in
- For each incoming data packet, do nonlocal product (light red region)



Note: Only point-to-point comm. needed!

Vector-vector dot product

Consider the Conjugate Gradient algorithm:

$\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0$

$\mathbf{p}_0 := \mathbf{r}_0$

$k := 0$

repeat

$\alpha_k := \dfrac{\mathbf{r}_k^{\mathrm{T}}\mathbf{r}_k}{\mathbf{p}_k^{\mathrm{T}}\mathbf{A}\mathbf{p}_k}$

$\mathbf{x}_{k+1} := \mathbf{x}_k + \alpha_k\mathbf{p}_k$

$\mathbf{r}_{k+1} := \mathbf{r}_k - \alpha_k\mathbf{A}\mathbf{p}_k$

if r_{k+1} is sufficiently small then exit loop

$\beta_k := \dfrac{\mathbf{r}_{k+1}^{\mathrm{T}}\mathbf{r}_{k+1}}{\mathbf{r}_k^{\mathrm{T}}\mathbf{r}_k}$

$\mathbf{p}_{k+1} := \mathbf{r}_{k+1} + \beta_k\mathbf{p}_k$

$k := k + 1$

end repeat

The result is $\mathbf{x}_{k+1}$

Source: Wikipedia

Vector-vector dot product

Consider the Conjugate Gradient algorithm:

$\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0$

$\mathbf{p}_0 := \mathbf{r}_0$

$k := 0$

repeat

$\alpha_k := \dfrac{\mathbf{r}_k^{\mathrm{T}}\mathbf{r}_k}{\mathbf{p}_k^{\mathrm{T}}\mathbf{A}\mathbf{p}_k}$

$\mathbf{x}_{k+1} := \mathbf{x}_k + \alpha_k \mathbf{p}_k$

$\mathbf{r}_{k+1} := \mathbf{r}_k - \alpha_k \mathbf{A}\mathbf{p}_k$

if r_{k+1} is sufficiently small then exit loop

$\beta_k := \dfrac{\mathbf{r}_{k+1}^{\mathrm{T}}\mathbf{r}_{k+1}}{\mathbf{r}_k^{\mathrm{T}}\mathbf{r}_k}$

$\mathbf{p}_{k+1} := \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k$

$k := k + 1$

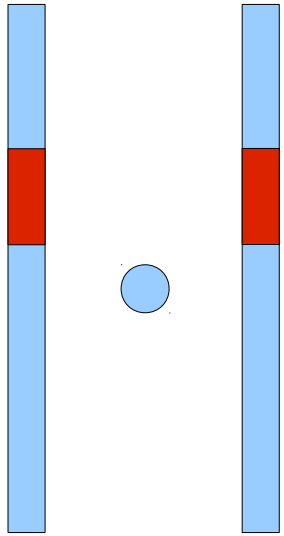
end repeat

The result is $\mathbf{x}_{k+1}$

Source: Wikipedia

Vector-vector dot product

Consider the dot product:



$$x \cdot y = \sum_{i=1}^{N} x_i y_i = \sum_{p=1}^{P} \left(\sum_{\text{local elements on proc } p} x_i y_i \right)$$

Vector-vector dot product

Consider the Conjugate Gradient algorithm:

- Implementation requires
  – 1 matrix-vector product
  – 2 vector-vector (dot) products
  per iteration

- Matrix-vector product can be done with point-to-point communication

- Dot-product requires global sum (reduction) and sending the sum to everyone (broadcast)

- On very large machines (1M+ cores), the global communication for the dot product becomes bottleneck!

Parallel preconditioners

Consider Krylov-space methods algorithm:

To solve $Ax=b$ we need

- Matrix-vector products $z=Ay$
- Various vector-vector operations
- A preconditioner $v=Pw$

- Want: P approximates A^{-1}

Question: What are the issues in parallel?

(For general considerations on preconditioners, see lectures 35-38.)

Parallel preconditioners

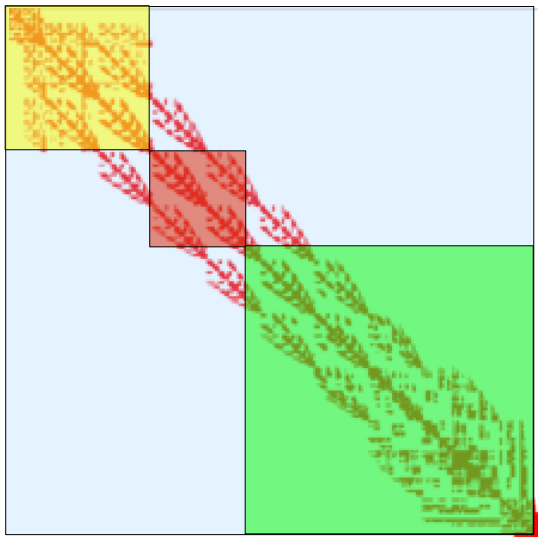
First idea: Block-diagonal preconditioners

Pros:

- *P* can be computed locally
- *P* can be applied locally (without communication)
- *P* can be approximated (SSOR, ILU on each block)

Cons:

- Deteriorates with larger numbers of processors
- Equivalent to Jacobi in the extreme of one row per processor

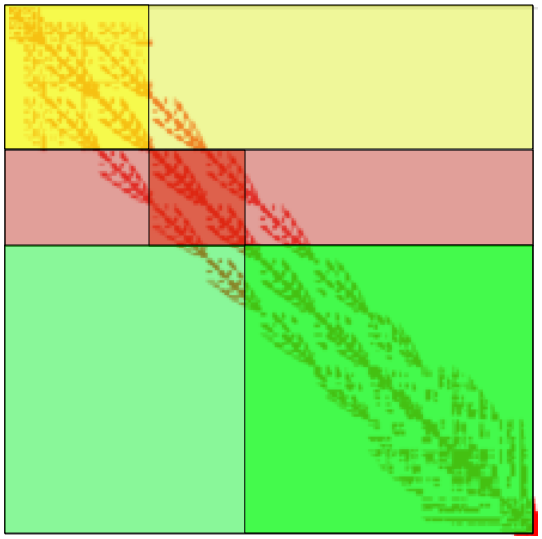


Lesson: Diagonal block preconditioners don't work well! We need data exchange!

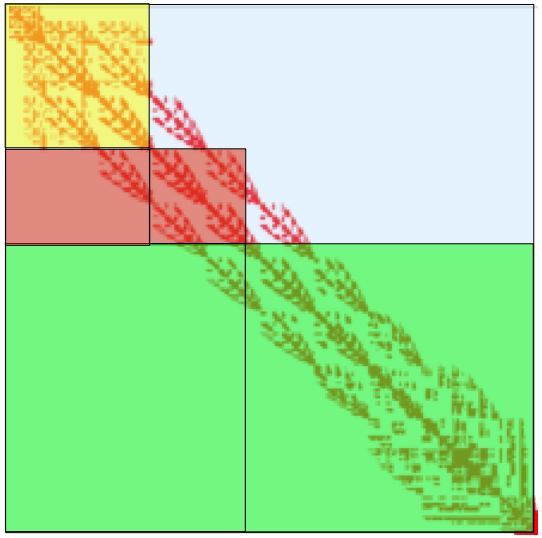
Parallel preconditioners

Second idea: Block-triangular preconditioners

Consider distributed storage of the matrix on 3 processors:

$A =$ 

Then form the preconditioner from the lower triangle of blocks:

$P^{-1} =$ 

Parallel preconditioners

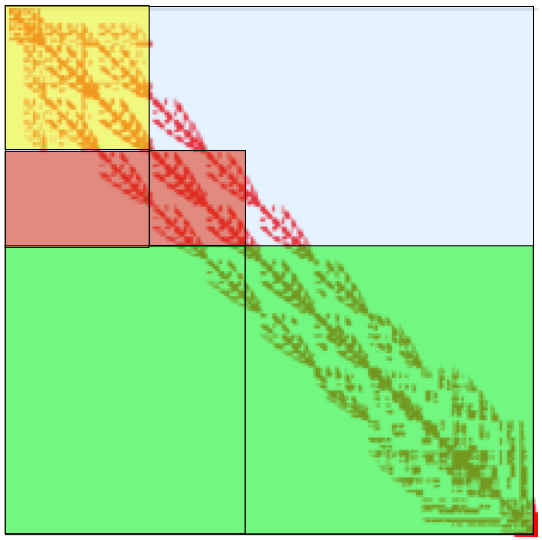
Second idea: Block-triangular preconditioners

Pros:

- *P* can be computed locally
- *P* can be applied locally
- *P* can be approximated (SSOR, ILU on each block)
- Works reasonably well

Cons:

- Equivalent to Gauss-Seidel in the extreme of one row per processor
- Is *sequential*!



Lesson: Data flow must have fewer then *O(#procs)* synchronization points!

Parallel preconditioners

What works:

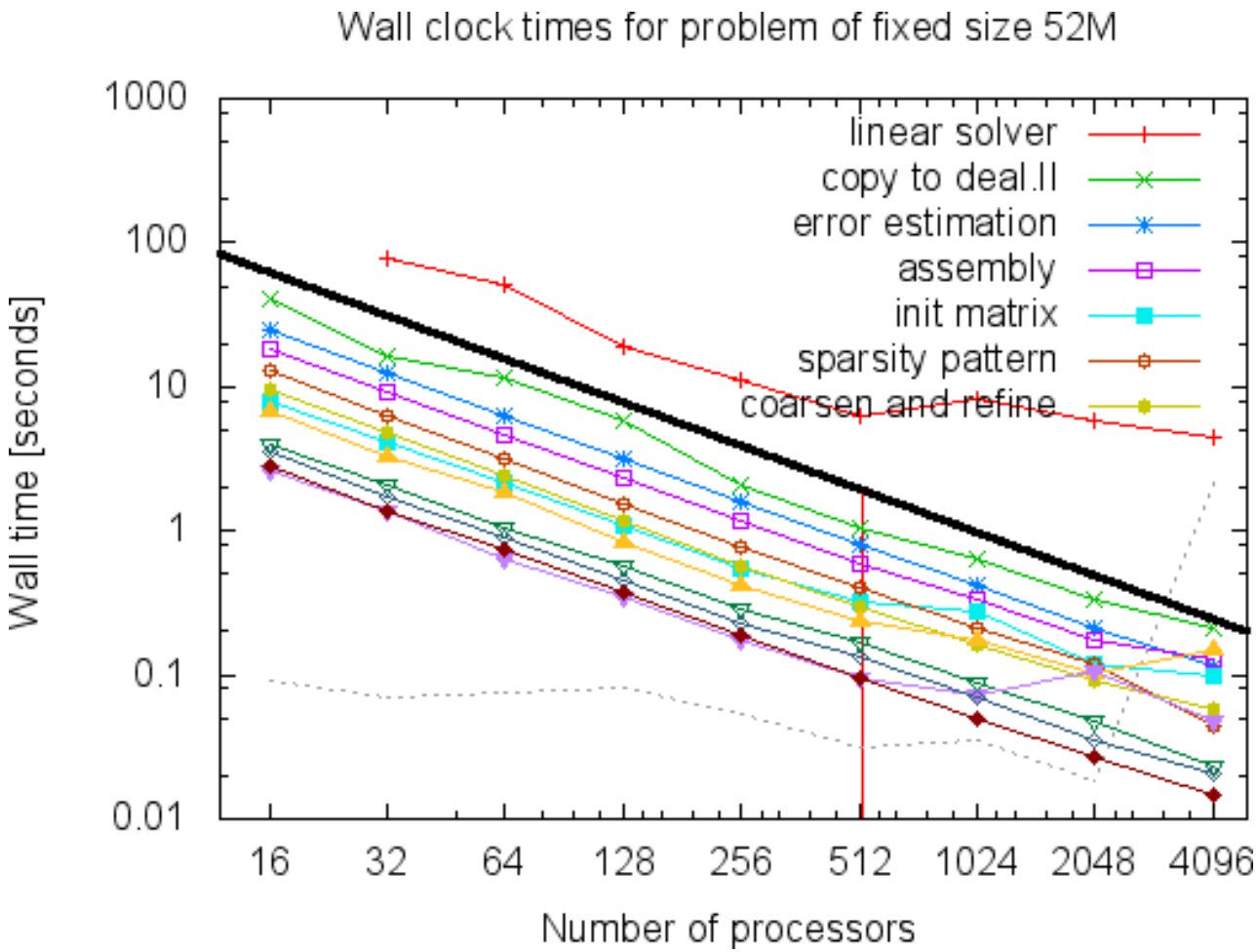
- Geometric multigrid methods for elliptic problems:
  - Require point-to-point communication in smoother
  - Very difficult to load balance with adaptive meshes
  - *O(N)* effort for overall solver

- Algebraic multigrid methods for elliptic problems:
  - Require point-to-point communication
    . in smoother
    . in construction of multilevel hierarchy
  - Difficult (but easier) to load balance
  - Not quite *O(N)* effort for overall solver

  - "Black box" implementations available (ML, hypre)

Parallel preconditioners

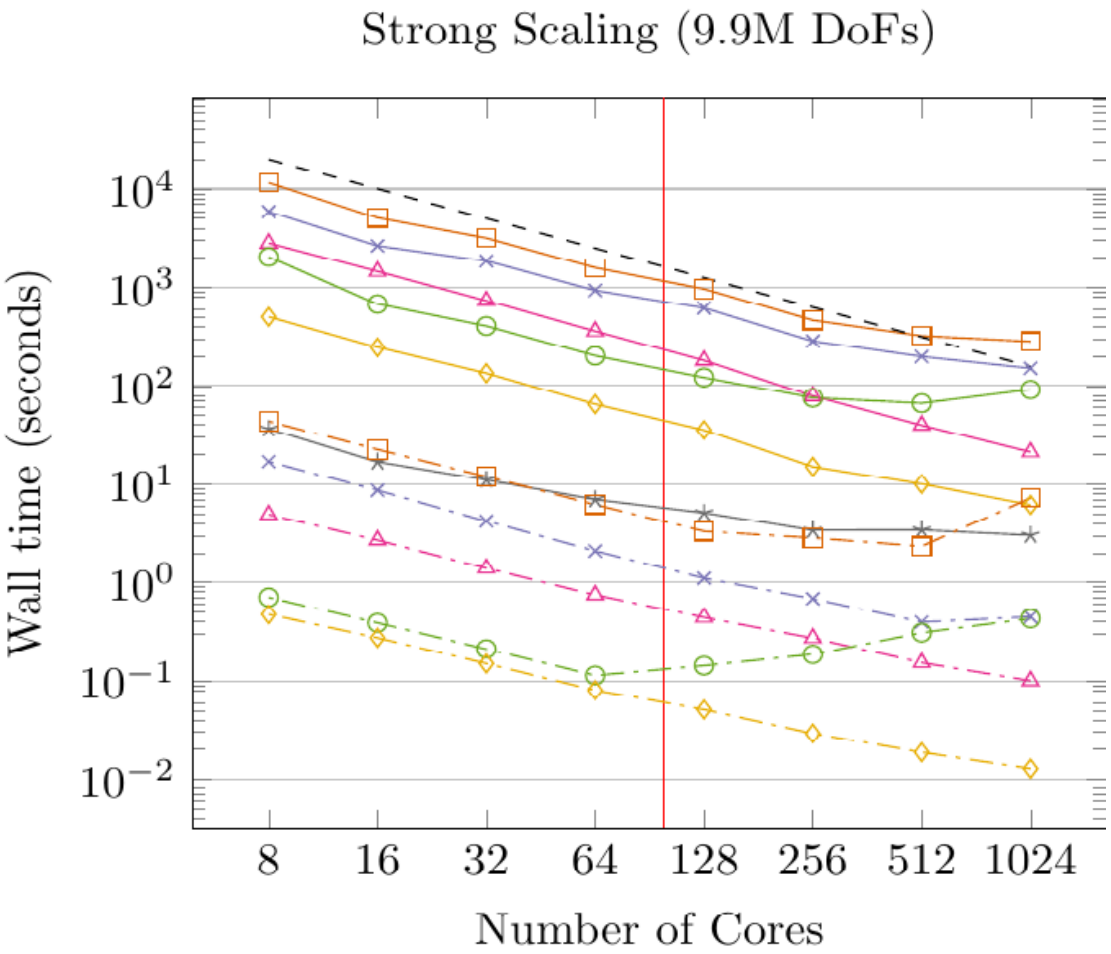
Examples (strong scaling):



Laplace equation (from Bangerth et al., 2011)

Parallel preconditioners

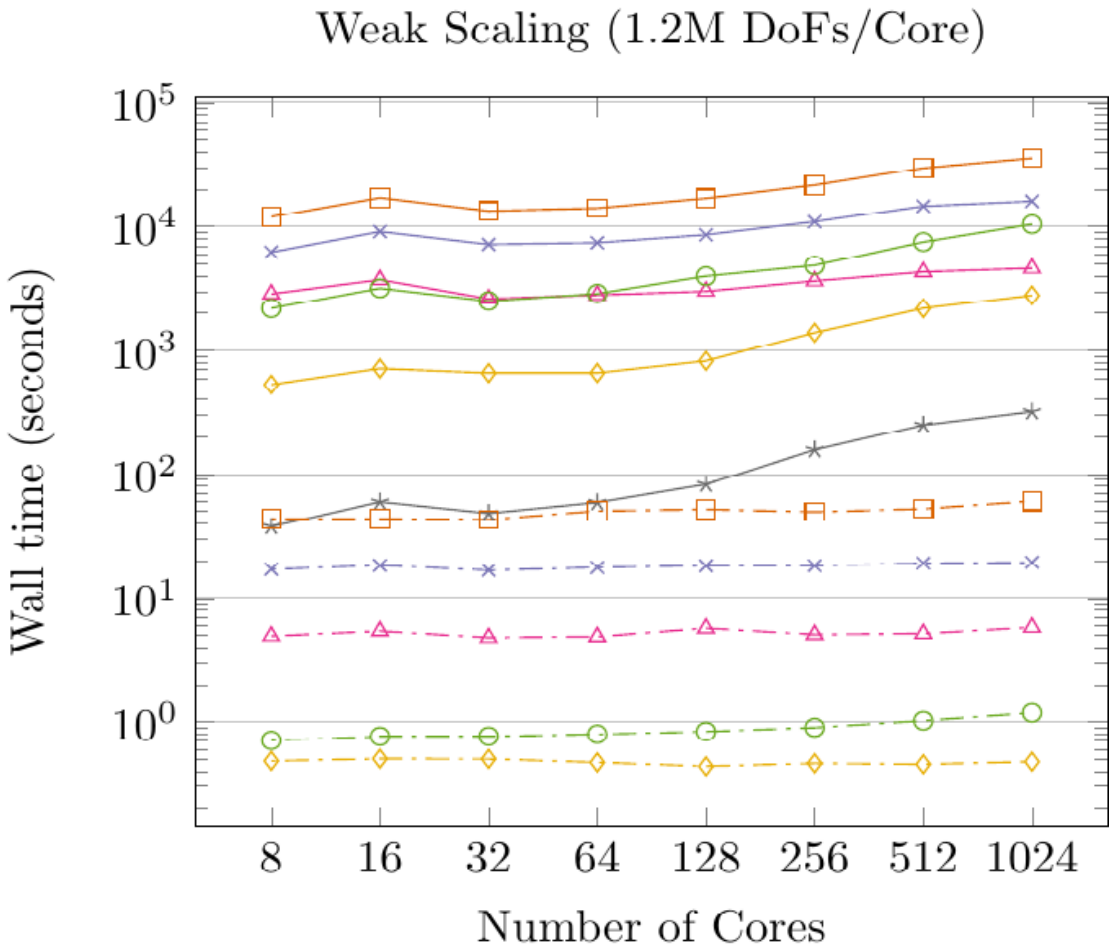
Examples (strong scaling):



Elasticity equation (from Frohne, Heister, Bangerth, submitted)

Parallel preconditioners

Examples (weak scaling):



Elasticity equation (from Frohne, Heister, Bangerth, submitted)

Parallel solvers

Summary:

- Mental model: See linear system as a large whole
- Apply Krylov-solver at the global level
- Use algebraic multigrid method (AMG) as black box preconditioner for elliptic blocks
- Build more complex preconditioners for block systems (see lecture 38)
- Might also try parallel direct solvers

MATH 676

–

Finite element methods in scientific computing

Wolfgang Bangerth, Texas A&M University