Finite element methods in scientific computing

Wolfgang Bangerth, Colorado State University

Lecture 1

Overview

The numerical solution of partial differential equations is an immensely practical field!

It requires us to know about:

- Partial differential equations
- Methods for discretizations, solvers, preconditioners
- Programming
- Adequate tools

Partial differential equations

Many of the big problems in scientific computing are described by partial differential equations (PDEs):

- Structural statics and dynamics
  – Bridges, roads, cars, …
- Fluid dynamics
  – Ships, pipe networks, …
- Aerodynamics
  – Cars, airplanes, rockets, …
- Plasma dynamics
  – Astrophysics, fusion energy
- But also in many other fields: Biology, finance, epidemiology, ...

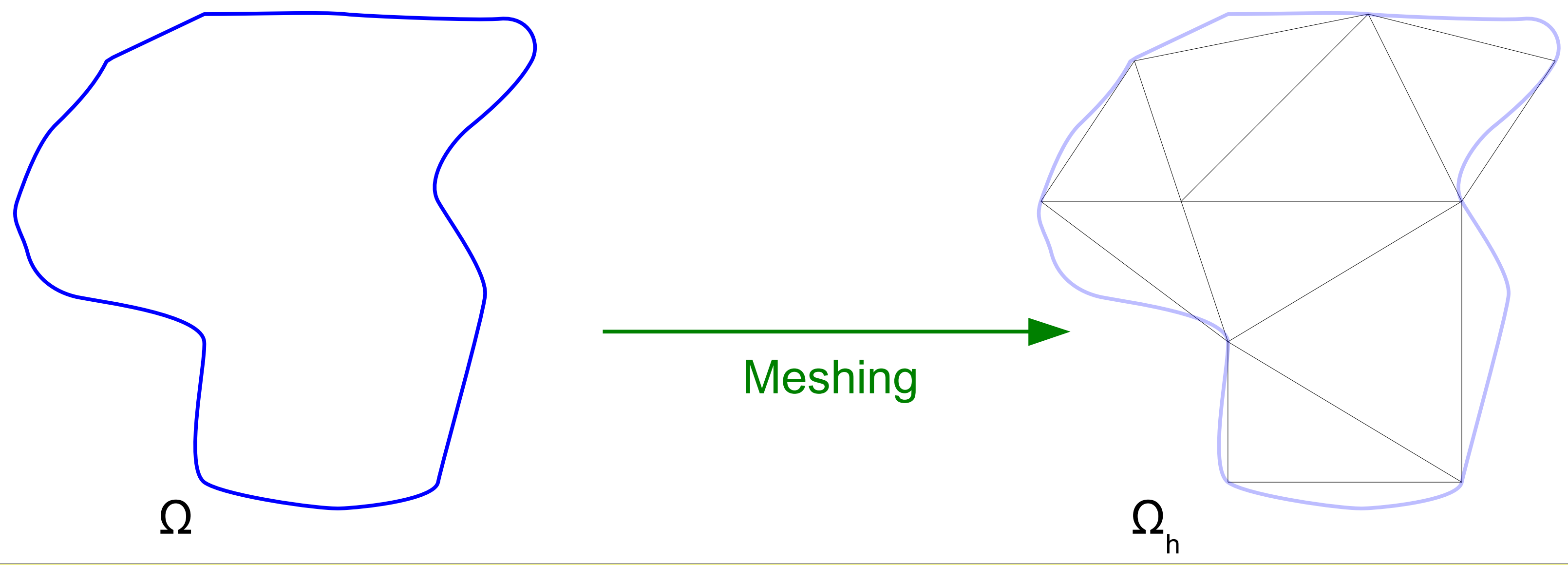
Numerics for PDEs

There are 3 standard tools for the numerical solution of PDEs:

- Finite element method (FEM)
- Finite volume method (FVM)
- Finite difference method (FDM)

Common features:

- Split the domain into small volumes (cells)



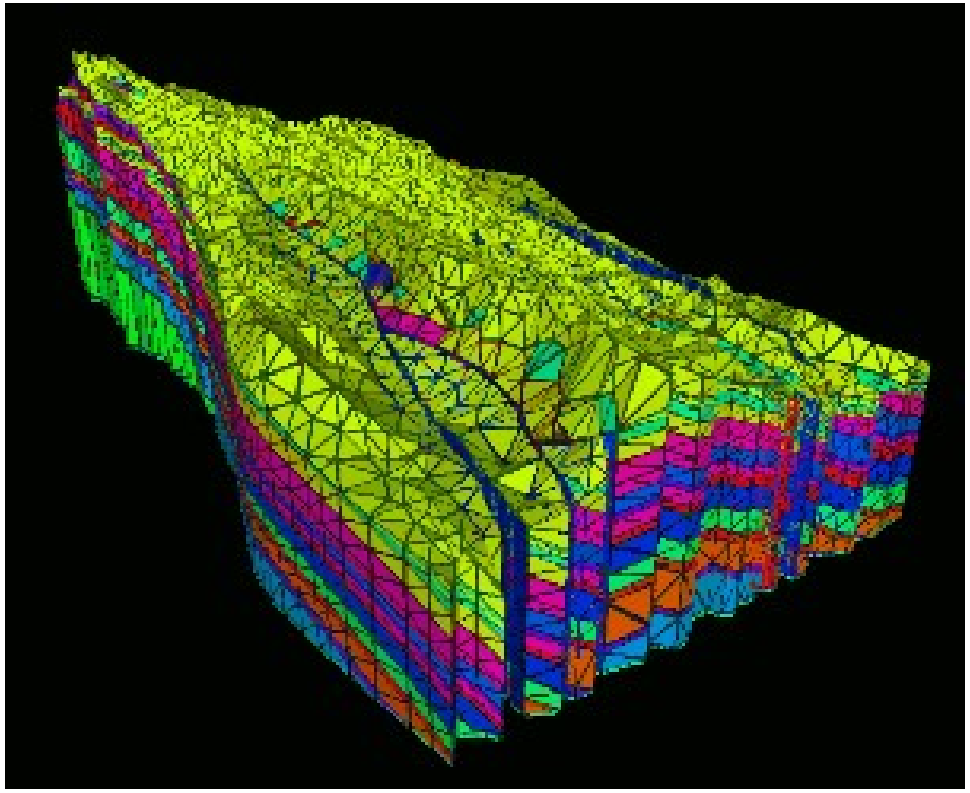
Numerics for PDEs

There are 3 standard tools for the numerical solution of PDEs:

- Finite element method (FEM)
- Finite volume method (FVM)
- Finite difference method (FDM)

Common features:

- Split the domain into small volumes (cells)



Numerics for PDEs

There are 3 standard tools for the numerical solution of PDEs:

- Finite element method (FEM)
- Finite volume method (FVM)
- Finite difference method (FDM)

Common features:

- Split the domain into small volumes (cells)
- Define balance relations on each cell
- Obtain and solve very large (non-)linear systems

Numerics for PDEs

There are 3 standard tools for the numerical solution of PDEs:

- Finite element method (FEM)
- Finite volume method (FVM)
- Finite difference method (FDM)

Common features:

- Split the domain into small volumes (cells)
- Define balance relations on each cell
- Obtain and solve very large (non-)linear systems

Today and tomorrow: We will not go into details of this, but consider only the parallel computing aspects.

Numerics for PDEs

Common features:

- Split the domain into small volumes (cells)
- Define balance relations on each cell
- Obtain and solve very large (non-)linear systems

Problems:

- Every code has to implement these steps
- There is only so much time in a day
- There is only so much expertise anyone can have

In addition:

- We don't just want a simple algorithm
- We want state-of-the-art methods for everything

Numerics for PDEs

Examples of what we would like to have:

- Adaptive meshes
- Realistic, complex geometries

- Quadratic or even higher order elements

- Multigrid solvers
- Scalability to 1000s of processors
- Efficient use of current hardware

- Graphical output suitable for high quality rendering

Q: How can we make all of this happen in a single code?

How we develop software

Q: How can we make all of this happen in a single code?

Not a question of feasibility but of how we develop software:

- Is every student developing their own software?
- Or are we re-using what others have done?

- Do we insist on implementing everything from scratch?
- Or do we build our software on existing libraries?

How we develop software

Q: How can we make all of this happen in a single code?

Not a question of feasibility but of how we develop software:

- Is every student developing their own software?
- Or are we re-using what others have done?

- Do we insist on implementing everything from scratch?
- Or do we build our software on existing libraries?

There has been a major shift on how we approach the second question in scientific computing over the past 10-15 years!

How we develop software

The secret to good scientific software is (re)using existing libraries!

Existing software

There is excellent software for almost every purpose!

Basic linear algebra (dense vectors, matrices):

- BLAS
- LAPACK

Parallel linear algebra (vectors, sparse matrices, solvers):

- PETSc
- Trilinos

Meshes, finite elements, etc:

- deal.II – the topic of this class
- …

Visualization, dealing with parameter files, ...

deal.II

deal.II is a finite element library. It provides:

- Meshes
- Finite elements, quadrature,
- Linear algebra
- Most everything you will ever need when writing a finite element code

On the web at

http://www.dealii.org/

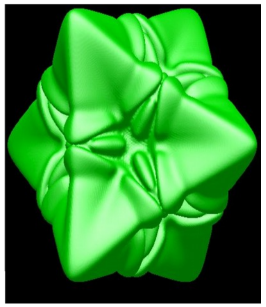
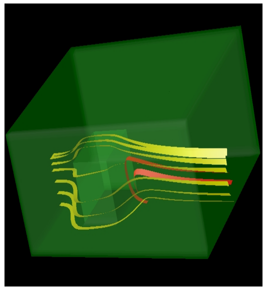
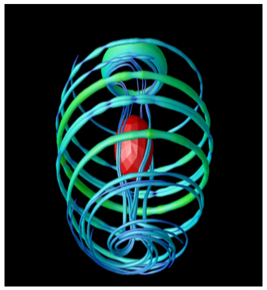
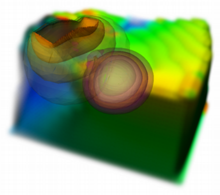
What's in deal.II

Linear algebra in deal.II:

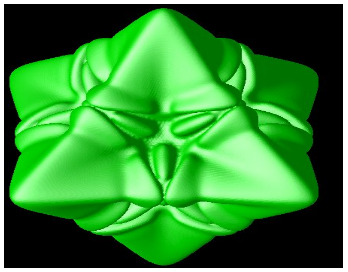
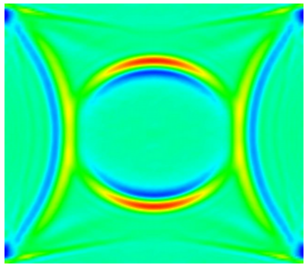
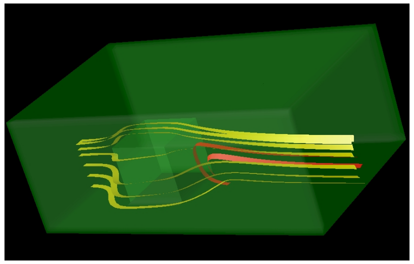
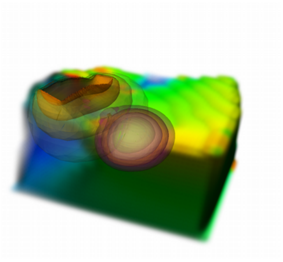
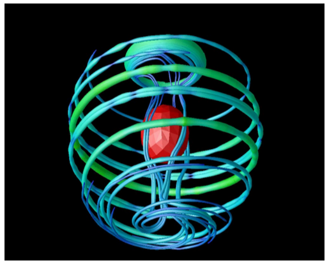
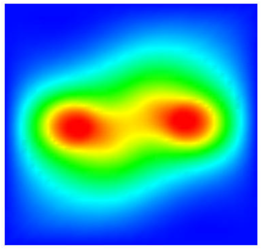
- Has its own sub-library for dense + sparse linear algebra
- Interfaces to PETSC, Trilinos, UMFPACK

Parallelization:

- Uses threads and tasks on multicore machines
- Uses MPI, up to 100,000s of processors

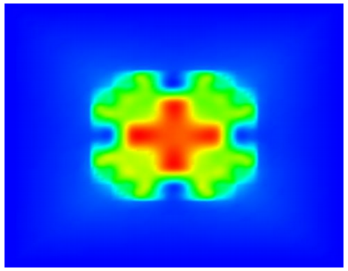
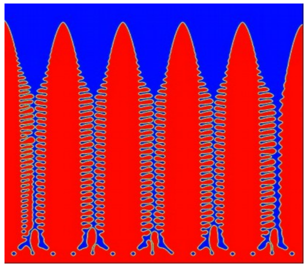
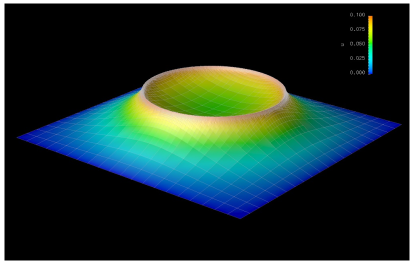
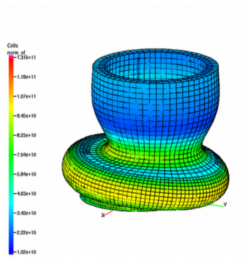
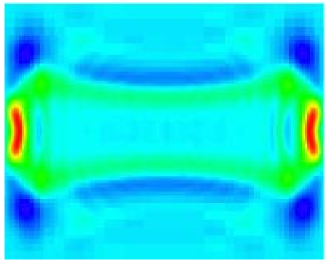
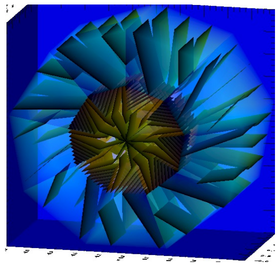





On the web

Visit the deal.II library:

http://www.dealii.org/

deal.II

- **Mission:**
  To provide everything that is needed in finite element computations.

- **Development:**
  As an open source project
  As an inviting community to all who want to contribute
  As professional-grade software to users

Lecture 2

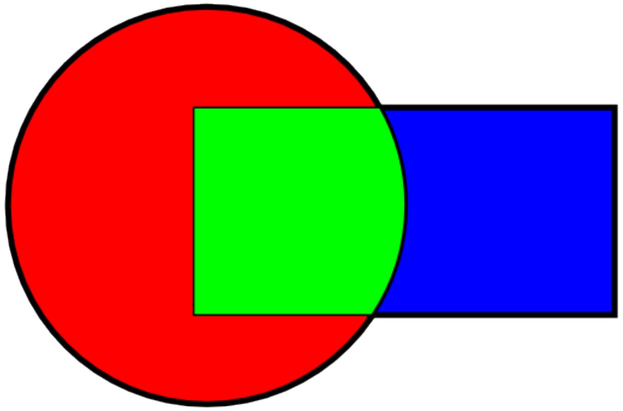
General approach to parallel solvers

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:

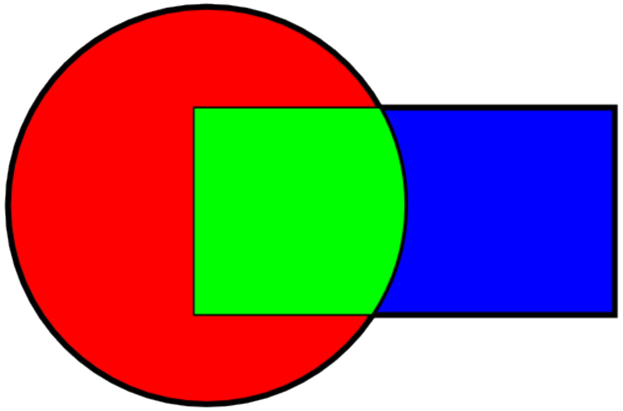


Source: Wikipedia

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:

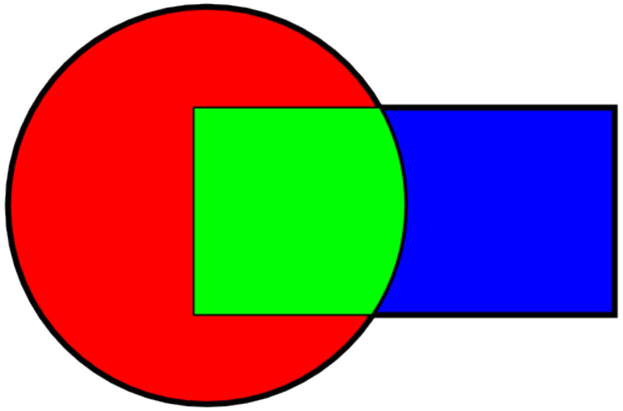


Approach (Hermann Schwarz, 1870):

- 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

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

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

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.

Lecture 3

Finite element methods with MPI

Philosophy:

- *Global objects* require *O(N)* memory (*N*=# of cells)
- *Every* global data structure needs to be distributed:
  - Triangulation
  - Constraints on the solution
  - Data attached to cells
  - Matrix
  - Solution and right hand side vectors
  - Postprocessed data (DataOut)
- **No processor may hold all data for a global object**

- Processors hold *O(N/P)* "locally owned" data
- Processors may also hold *O(εN/P)* "ghost elements"

Finite element methods with MPI

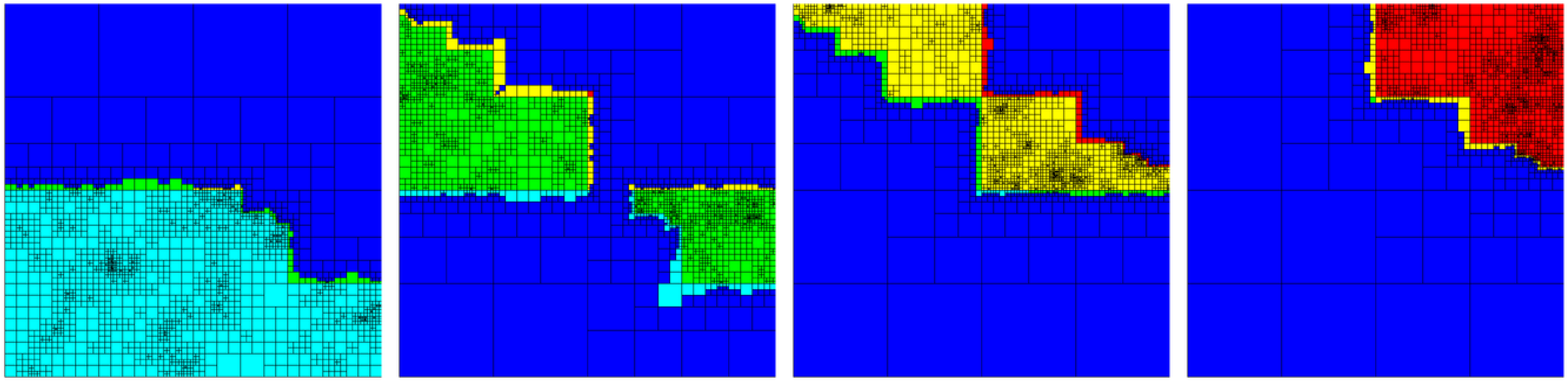
Philosophy:

- Every processor may only work on locally owned data (possibly using ghost data as necessary)
- Software must carefully communicate data that may be necessary early on, try to avoid further communication
- Use PETSc/Trilinos for linear algebra
- (Almost) No handwritten MPI necessary in user code

Finite element methods with MPI

Example:

- There is an “abstract”, global triangulation
- Each processor has a triangulation object that stores “locally owned”, “ghost” and “artificial” cells (and that's all it knows):



P=0 *P=1* *P=2* *P=3*

(magenta, green, yellow, red: cells owned by processors 0, 1, 2, 3; blue: artificial cells)

Parallel user programs

How user programs need to be modified for parallel computations:

- Need to let
  – system matrix, vectors
  – hanging node constraints
  know about what is *locally owned*, *locally relevant*

- Need to restrict work to locally owned data
  Communicate everything else on an *as-needed basis*
- Need to create one output file per processor

- Everything else can happen in libraries under the hood

An MPI example: MatVec

Situation:

- Multiply a large *NxN* matrix by a vector of size *N*
- Matrix is assumed to be dense

- Every one of *P* processors stores *N/P* rows of the matrix
- Every processor stores *N/P* elements of each vector

- For simplicity: *N* is a multiple of *P*

An MPI example: MatVec

```
struct ParallelVector {
    unsigned int size;
    unsigned int my_elements_begin;
    unsigned int my_elements_end;
    double *elements;

    ParallelVector (unsigned int sz,MPI_Comm comm) {
        size = sz;
        int comm_size, my_rank;
        MPI_Comm_size (comm, &comm_size);
        MPI_Comm_rank (comm, &my_rank);
        my_elements_begin = size/comm_size*my_rank;
        my_elements_end = size/comm_size*(my_rank+1);
        elements = new double[my_elements_end-my_elements_begin];
    }
};
```

An MPI example: MatVec

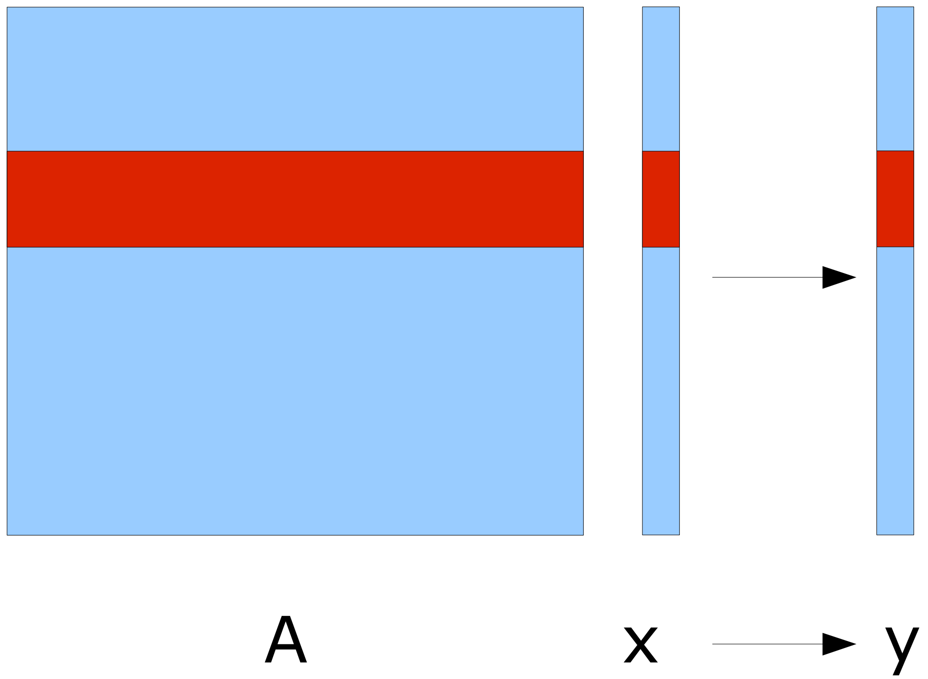
```
struct ParallelSquareMatrix {
    unsigned int size;
    unsigned int my_rows_begin;
    unsigned int my_rows_end;
    double *elements;

    ParallelSquareMatrix (unsigned int sz,MPI_Comm comm) {
        size = sz;
        int comm_size, my_rank;
        MPI_Comm_size (comm, &comm_size);
        MPI_Comm_rank (comm, &my_rank);
        my_rows_begin = size/comm_size*my_rank;
        my_rows_end = size/comm_size*(my_rank+1);
        elements = new double[(my_rows_end-my_rows_begin)*size];
    }
};
```

An MPI example: MatVec

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*

An MPI example: MatVec

```
void mat_vec (A, x, y) {
    int comm_size=..., my_rank=...;
    for (row_block=0; row_block<comm_size; ++row_block)
       if (row_block == my_rank) {
          for (col_block=0; col_block<comm_size; ++col_block)
             if (col_block == my_rank) {
                for (i=A.my_rows_begin; i<A.my_rows_end; ++i)
                  for (j=A.size/comm_size*col_block; ...)
                     y.elements[i-y.my_rows_begin] = A[...i,j...] * x[...j...];
             } else {
                double *tmp = new double[A.size/comm_size];
                MPI_Recv (tmp, …, row_block, …);
                for (i=A.my_rows_begin; i<A.my_rows_end; ++i)
                  for (j=A.size/comm_size*col_block; ...)
                     y.elements[i-y.my_rows_begin] = A[...i,j...] * tmp[...j...];
                delete tmp;
             }
       } else {
          MPI_Send (x.elements, …, row_block, …);
       }
}
```

An MPI example: MatVec

Analysis of this algorithm

- We only send data right when we need it:
  – receiving processor has to wait
  – has nothing to do in the meantime
  A better algorithm would:
  – send out its data to all other processors
  – receive messages as needed (maybe already here)
- As a general rule:
  – send data as soon as possible
  – receive it as late as possible
  – try to interleave computations between sends/receives
- We repeatedly allocate/deallocate memory – should set up buffer only once

An MPI example: MatVec

```
void vmult (A, x, y) {
     int comm_size=..., my_rank=...;
     for (row_block=0; row_block<comm_size; ++row_block)
        if (row_block != my_rank)
           MPI_Send (x.elements, …, row_block, …);

    col_block = my_rank;
    for (i=A.my_rows_begin; i<A.my_rows_end; ++i)
       for (j=A.size/comm_size*col_block; ...)
          y.elements[i-y.my_rows_begin] = A[...i,j...] * x[...j...];

    double *tmp = new double[A.size/comm_size];
     for (col_block=0; col_block<comm_size; ++col_block)
        if (col_block != my_rank) {
           MPI_Recv (tmp, …, row_block, …);
           for (i=A.my_rows_begin; i<A.my_rows_end; ++i)
                for (j=A.size/comm_size*col_block; ...)
                    y.elements[i-y.my_rows_begin] = A[...i,j...] * tmp[...j...];
        }
     delete tmp;
}
```

Message Passing Interface (MPI)

Notes on using MPI:

- Usually, algorithms need data that resides elsewhere
- Communication needed

- Distributed computing lives in the conflict zone between
  - trying to keep as much data available locally to avoid communication
  - not creating a memory/CPU bottleneck

- MPI makes the flow of information explicit
- Forces programmer to design data structures/algorithms for communication

- Well written programs have relatively few MPI calls

Lecture 4

Solver questions

The finite element method provides us with a linear system

$$Ax = b$$

We know:

- *A* is *large*: typically a few 1,000 up to a few billions
- *A* is *sparse*: typically no more than a few 100 entries per row
- *A* is typically *ill-conditioned*: condition numbers up to 10^9

Question:

How do we go about solving such linear systems?

Direct solvers

Direct solvers – compute a decomposition of *A*:

- Can be thought of as variant of LU decomposition that finds triangular factors *L, U* so that

$$A = LU$$

- *Sparse direct* solvers save memory and CPU time by considering the sparsity pattern of *A*
- Very robust
- Work grows as $O(N^{1+2(d-1)/d})$, i.e.,
  – $O(N^2)$ in 2d
  – $O(N^{7/3})$ in 3d
- Memory grows as $O(N^{1+(d-1)/d})$, i.e.,
  – $O(N^{3/2})$ in 2d
  – $O(N^{5/3})$ in 3d

Direct solvers

Where to get a direct solver:

- Several very high quality, open source packages
- Most widely used ones are
  - UMFPACK
  - SuperLU
  - MUMPS
- The latter two are even parallelized

But:

It is generally very difficult to implement direct solvers efficiently in parallel.

Iterative solvers

Iterative solvers improve the solution in each iteration:

- Start with an initial guess x_0
- Continue iterations till a stopping criterion is satisfied (typically that the error/residual is less than a tolerance)
- Return final guess x_k

- Depending on solver and preconditioner type, work can be *O(N)* or (much) worse
- Memory is typically linear, i.e., *O(N)*

Note: The final guess does not solve $Ax=b$ exactly!

Iterative solvers

There is a wide variety of iterative solvers:

- CG, MinRes, GMRES, ...
- All of them are actually rather simple to implement: They usually need less than 200 lines of code
- Consequently, many high quality implementations

Advantage: Only need multiplication with the matrix, no modification/insertion of matrix elements required.

Disadvantage: Efficiency hinges on availability of good preconditioners.

Direct vs iterative

Guidelines for direct solvers vs iterative solvers:

Direct solvers:

- ✓ *Always* work, for any invertible matrix
- ✓ Faster for problems with <100k unknowns
- ✗ Need too much memory + CPU time for larger problems
- ✗ **Do not parallelize well**

Iterative solvers:

- ✓ Need *O(N)* memory
- ✓ Can solve *very* large problems
- ✓ **Often parallelize well**
- ✗ Choice of solver/preconditioner depends on problem

Advice for iterative solvers

There is a wide variety of iterative solvers:

CG:	Conjugate gradients
MinRes:	Minimal residuals
GMRES:	Generalized minimal residuals
F-GMRES:	Flexible GMRES
SymmLQ:	Symmetric LQ decomposition
BiCGStab:	Biconjugate gradients stabilized
QMR:	Quasi-minimal residual
TF-QMR:	Transpose-free QMR
…	

Which solver to choose depends on the properties of the matrix, primarily *symmetry* and *definiteness*!

Advice for iterative solvers

Guidelines for use:

• CG:	Matrix is symmetric, positive definite
• MinRes:	–
• GMRES:	Catch-all
• F-GMRES:	Catch-all with variable preconditioners
• SymmLQ:	–
• BiCGStab:	Matrix is non-symmetric but positive definite
• QMR:	–
• TF-QMR:	–
• All others:	–

In reality, only CG, BiCGStab and (F-)GMRES are used much.

Advice for iterative solvers

Note:

All iterative solvers are bad without a good preconditioner!

The art of devising a good iterative solver is to devise a good preconditioner!

Lecture 5

Observations on iterative solvers

The finite element method provides us with a linear system

$$Ax = b$$

that we then need to solve.

Basic observations:

- For sparse direct solvers, speed of solution only depends on sparsity pattern
- For iterative solvers, performance also depends on the *values* in A
- Performance measures:
  – number of iterations
  – cost of every iteration

Observations on iterative solvers

The finite element method provides us with a linear system

$$Ax = b$$

that we then need to solve.

Factors affecting performance of iterative solvers:

- Symmetry of a matrix
- Whether *A* is definite
- Condition number of *A*
- How the eigenvalues of *A* are clustered

- Whether *A* is reducible/irreducible

Observations on iterative solvers

Example 1: Using CG to solve

$$Ax = b$$

where *A* is SPD, each iteration reduces the residual by a factor of

$$r = \frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1} < 1$$

- For a tolerance ε we need $n = \frac{\log \epsilon}{\log r}$ iterations
- **Problem:** The condition number typically grows with the problem size $\rightarrow$ number of iterations grows

Observations on iterative solvers

Example 2: When solving

$$Ax = b$$

where *A* has the form

$$A = \begin{pmatrix} a_{11} & 0 & 0 & \cdots \\ 0 & a_{22} & 0 & \cdots \\ 0 & 0 & a_{33} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

then every decent iterative solver converges in 1 iteration.

Note 1: This, even though condition number may be large
Note 2: This is true, in particular, if $A=I$.

The idea of preconditioners

Idea: When solving

$$Ax = b$$

maybe we can find a matrix P^{-1} and instead solve

$$P^{-1}Ax = P^{-1}b$$

Observation 1: If $P^{-1}A \sim D$ then solving should require less iterations

Corollary: The perfect preconditioner is a multiple of the inverse matrix, i.e., $P^{-1}=A^{-1}$.

The idea of preconditioners

Idea: When solving

$$A x = b$$

maybe we can find a matrix P^{-1} and instead solve

$$P^{-1} A x = P^{-1} b$$

Observation 2: Iterative solvers only need matrix-vector multiplications, no element-by-element access.

Corollary: It is sufficient if P^{-1} is just an operator

The idea of preconditioners

Idea: When solving

$$Ax = b$$

maybe we can find a matrix P^{-1} and instead solve

$$P^{-1}Ax = P^{-1}b$$

Observation 3: There is a tradeoff:

fewer iterations vs *cost of preconditioner.*

Corollary: Preconditioning only works if P^{-1} is cheap to compute and if P^{-1} is cheap to *apply* to a vector.

Consequence: $P^{-1}=A^{-1}$ does not qualify.

The idea of preconditioners

Notes on the following lectures:

- For quantitative analysis, one typically needs to consider the *spectrum* of operators and preconditioners

- Here, the goal is simply to get an “intuition” on how preconditioners work

Lecture 6

Constructing preconditioners

Remember: When solving the preconditioned system

$$P^{-1} A x = P^{-1} b$$

then the best preconditioner is $P^{-1} = A^{-1}$.

Problem: (i) We can't compute it efficiently. (ii) If we could, we would not need an iterative solver.

But: Maybe we can approximate $P^{-1} \sim A^{-1}$.

Idea 1: Do we know of other iterative solution techniques?

Idea 2: Use incomplete decompositions.

Constructing preconditioners

Approach 1: Remember the oldest iterative techniques!

To solve $Ax = b$ we can use *defect correction*:

- Under certain conditions, the iteration:

$$x^{(k+1)} = x^{(k)} - P^{-1}(Ax^{(k)} - b)$$

  will converge to the exact solution *x*

- Unlike Krylov-space methods, convergence is linear
- The best preconditioner is again $P^{-1} \sim A^{-1}$

Constructing preconditioners

Approach 1: Remember the oldest iterative techniques!

Preconditioned defect correction for $Ax = b, \quad A = L+D+U$:

- Jacobi iteration:

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

- The Jacobi preconditioner is then

$$P^{-1} = \omega D^{-1}$$

which is easy to compute and apply.

Note: We don't need the scaling (“relaxation”) factor.

Constructing preconditioners

Approach 1: Remember the oldest iterative techniques!

Preconditioned defect correction for $Ax = b, \qquad A = L+D+U$:

- Gauss-Seidel iteration:

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

- The Gauss-Seidel preconditioner is then

$$P^{-1} = \omega (L+D)^{-1} \qquad \text{i.e. } h = P^{-1} r \text{ solves } (L+D) h = \omega r$$

  which is easy to compute and apply as *L+D* is triangular.

Note 1: We don't need the scaling (“relaxation”) factor.
Note 2: This preconditioner is not symmetric.

Constructing preconditioners

Approach 1: Remember the oldest iterative techniques!

Preconditioned defect correction for $Ax = b, \qquad A = L+D+U$:

- SOR (Successive Over-Relaxation) iteration:

$$x^{(k+1)} = x^{(k)} - \omega(D+\omega L)^{-1}(Ax^{(k)}-b)$$

- The SOR preconditioner is then

$$P^{-1} = (D+\omega L)^{-1}$$

Note 1: This preconditioner is not symmetric.
Note 2: We again don't care about the constant factor in *P*.

Constructing preconditioners

Approach 1: Remember the oldest iterative techniques!

Preconditioned defect correction for $Ax = b, \qquad A = L+D+U$:

- SSOR (Symmetric Successive Over-Relaxation) iteration:

$$x^{(k+1)} = x^{(k)} - \frac{1}{\omega(2-\omega)}(D+\omega U)^{-1} D (D+\omega L)^{-1}(Ax^{(k)}-b)$$

- The SSOR preconditioner is then

$$P^{-1} = (D+\omega U)^{-1} D (D+\omega L)^{-1}$$

Note: This preconditioner is now symmetric if *A* is symmetric!

Constructing preconditioners

Approach 1: Remember the oldest iterative techniques!

Common observations about preconditioners from stationary iterations:

- Have been around for a long time
- Generally useful for small problems (<100,000 DoFs)
- Not particularly useful for larger problems

Constructing preconditioners

Approach 2: Approximations to A^{-1}

Idea 1: Incomplete decompositions

- Incomplete LU (ILU):
  Perform an LU decomposition on A but only keep elements of L, U that fit into the sparsity pattern of A

- Incomplete Cholesky (IC):
  LL^T decomposition if A is symmetric

- Many variants:
  – strengthen diagonal
  – augment sparsity pattern
  – thresholding of small/large elements

Summary

Conceptually: We now need to solve the linear system

$$P^{-1} A x = P^{-1} b$$

Goal: We would like to approximate $P^{-1} \sim A^{-1}$.

But: We don't need to know the entries of P^{-1} – we only see it as an operator.

Then: We can put it all into an iterative solver such as Conjugate Gradients that only requires matrix-vector products.

Lecture 7

Global solvers

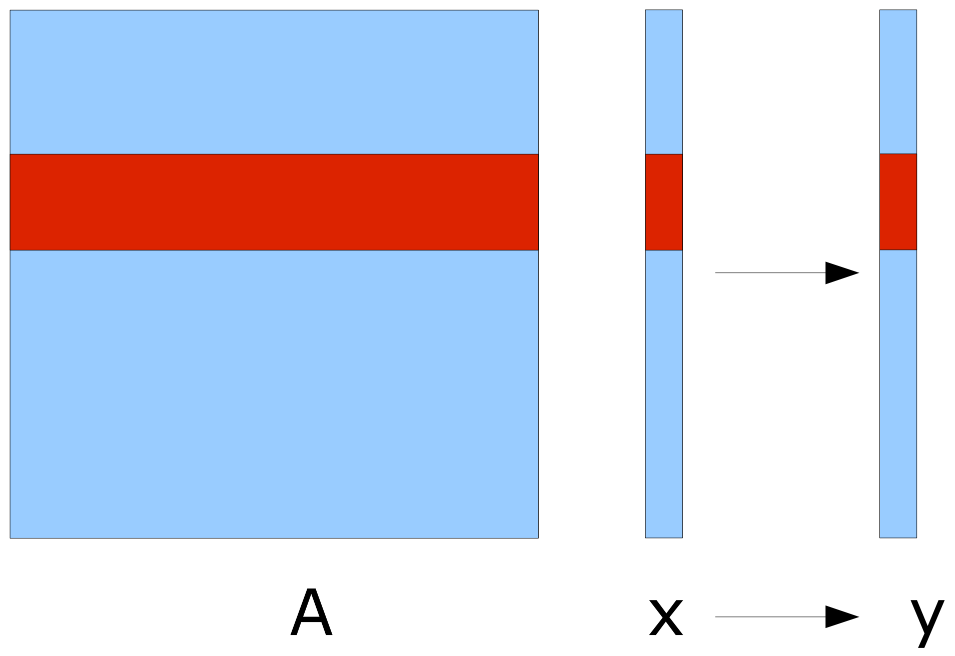
Examples for a few necessary steps:

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

Matrix-vector product

What does processor *P* need:

- Graphical representation of what *P* owns:

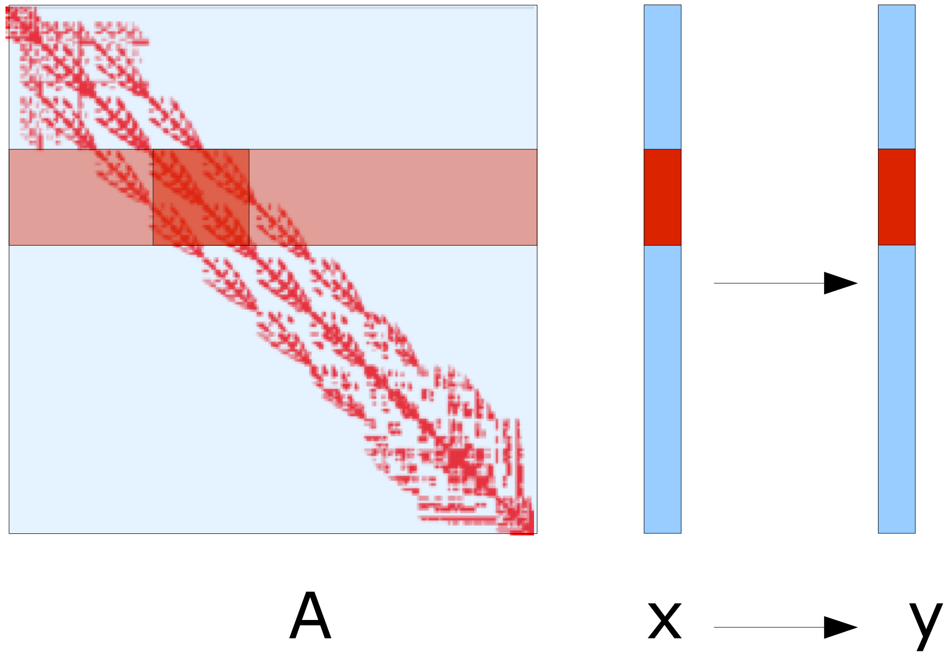


- To compute the *locally owned* elements of *y*, processor *P* needs **all** elements of *x*
- All processors need to send their share of *x* to everyone

Matrix-vector product

What does processor *P* need:

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

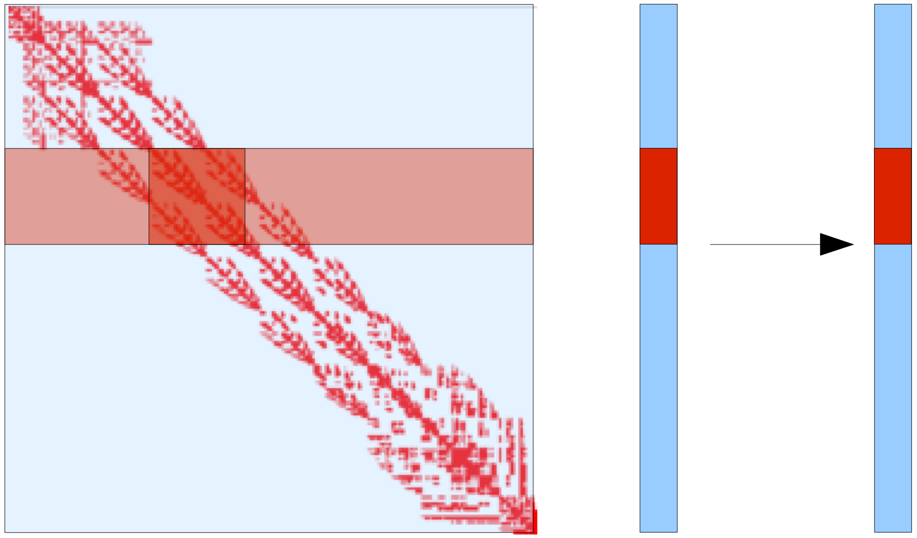


For the *locally owned* elements of *y*, processor *P* needs **all** x_j for which there is a nonzero A_{ij} for a locally owned row *i*.

Matrix-vector product

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

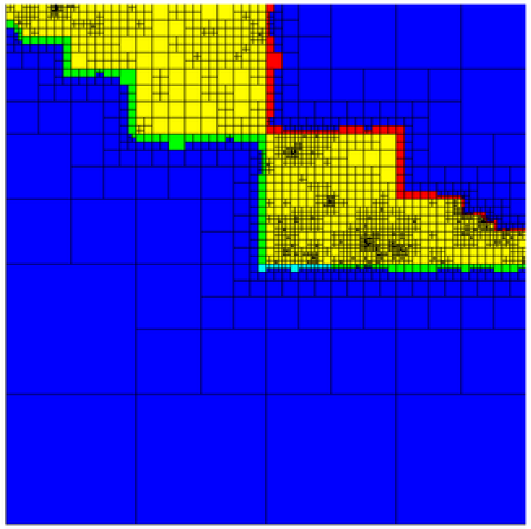
- All elements x_j for which there is a nonzero A_{ij} for a locally owned row *i*.
- In other words, if x_i is a locally owned DoF, we need all x_j that couple with x_i
- These are exactly the *locally relevant degrees of freedom*
- They live on *ghost cells*



Matrix-vector product

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

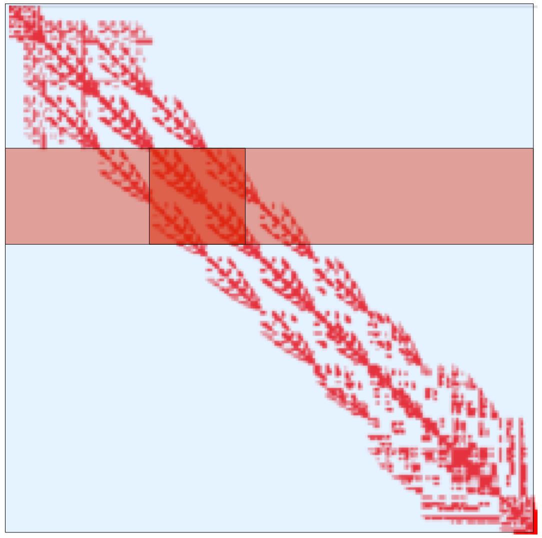
- All elements x_j for which there is a nonzero A_{ij} for a locally owned row *i*.
- In other words, if x_i is a locally owned DoF, we need all x_j that couple with x_i
- These are exactly the *locally relevant degrees of freedom*
- They live on *ghost cells*



Matrix-vector product

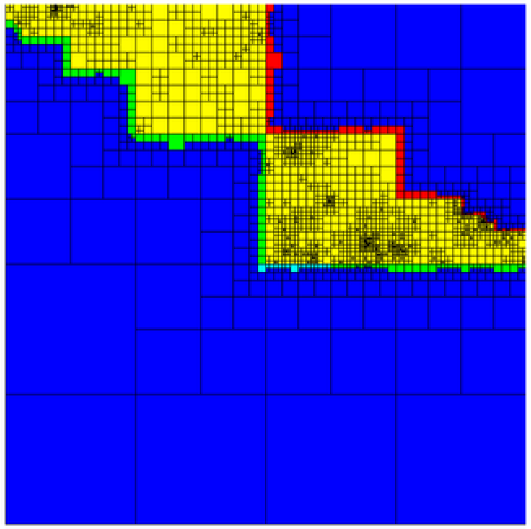
Parallel matrix-vector products for sparse matrices:

- 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
    $\alpha_k := \frac{\mathbf{r}_k^{\mathrm{T}}\mathbf{r}_k}{\mathbf{p}_k^{\mathrm{T}}\mathbf{A}\mathbf{p}_k}$
    $\mathbf{x}_{k+1} := \mathbf{x}_k + \alpha_k\mathbf{p}_k$
    $\mathbf{r}_{k+1} := \mathbf{r}_k - \alpha_k\mathbf{A}\mathbf{p}_k$
    if $r_{k+1}$ is sufficiently small then exit loop
    $\beta_k := \frac{\mathbf{r}_{k+1}^{\mathrm{T}}\mathbf{r}_{k+1}}{\mathbf{r}_k^{\mathrm{T}}\mathbf{r}_k}$
    $\mathbf{p}_{k+1} := \mathbf{r}_{k+1} + \beta_k\mathbf{p}_k$
    $k := k + 1$
end repeat
The result is $\mathbf{x}_{k+1}$
```

Source: Wikipedia

Vector-vector dot product

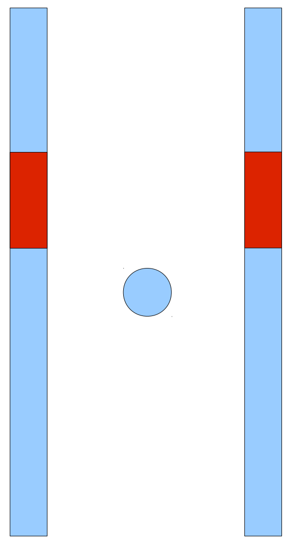
Consider the Conjugate Gradient algorithm:

```
$\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0$
$\mathbf{p}_0 := \mathbf{r}_0$
$k := 0$
repeat
    $\alpha_k := \frac{\mathbf{r}_k^{\mathrm{T}}\mathbf{r}_k}{\mathbf{p}_k^{\mathrm{T}}\mathbf{A}\mathbf{p}_k}$
    $\mathbf{x}_{k+1} := \mathbf{x}_k + \alpha_k\mathbf{p}_k$
    $\mathbf{r}_{k+1} := \mathbf{r}_k - \alpha_k\mathbf{A}\mathbf{p}_k$
    if $r_{k+1}$ is sufficiently small then exit loop
    $\beta_k := \frac{\mathbf{r}_{k+1}^{\mathrm{T}}\mathbf{r}_{k+1}}{\mathbf{r}_k^{\mathrm{T}}\mathbf{r}_k}$
    $\mathbf{p}_{k+1} := \mathbf{r}_{k+1} + \beta_k\mathbf{p}_k$
    $k := k + 1$
end repeat
The result is $\mathbf{x}_{k+1}$
```

Source: Wikipedia

Vector-vector dot product

Consider the dot product:



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

Parallel considerations

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)
- All of this is easily doable in a parallel code

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}

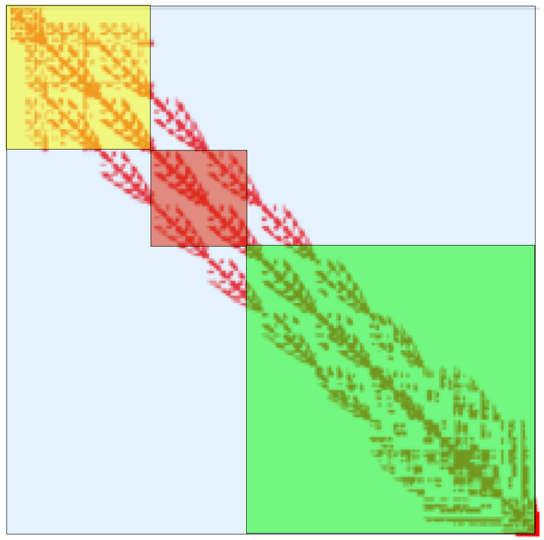
Question: What are the issues in parallel?

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

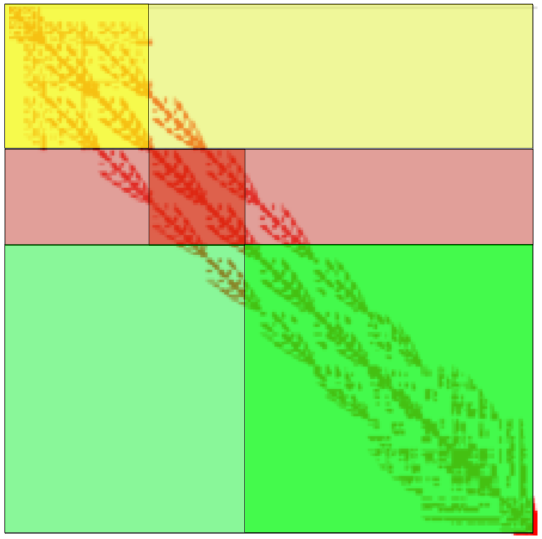


Lesson: Diagonal block preconditioners don't work well! We need data exchange!

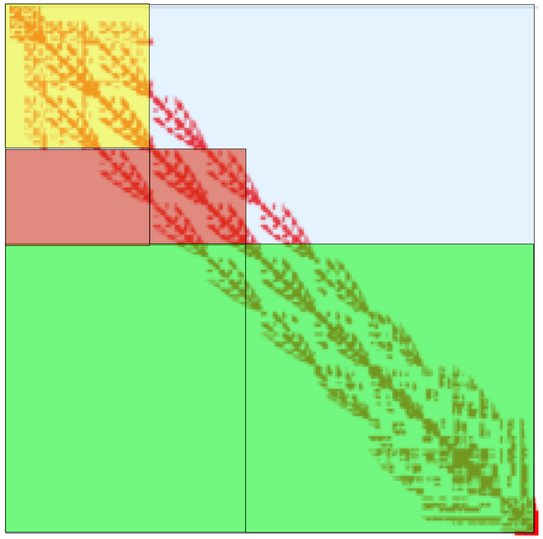
Parallel preconditioners

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

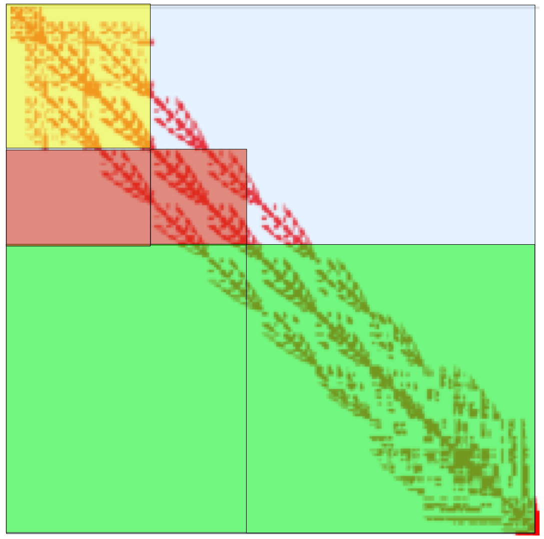
Second idea: Block-triangular preconditioners

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:

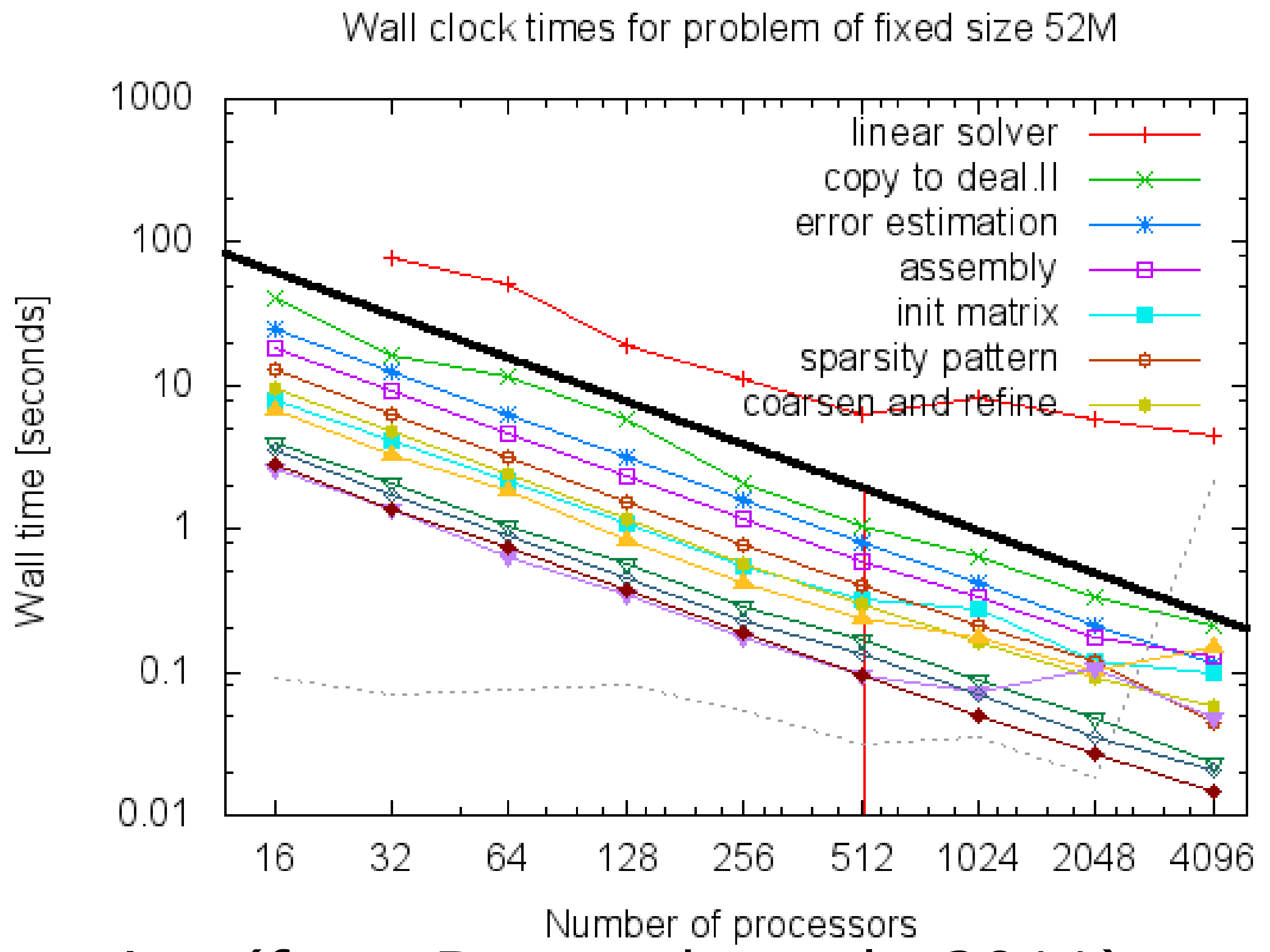
- Geometric multigrid methods for elliptic problems:
  – Require point-to-point communication in smoother
  – Very difficult to load balance with adaptive meshes
  – *O(N)* effort for overall solver

- 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

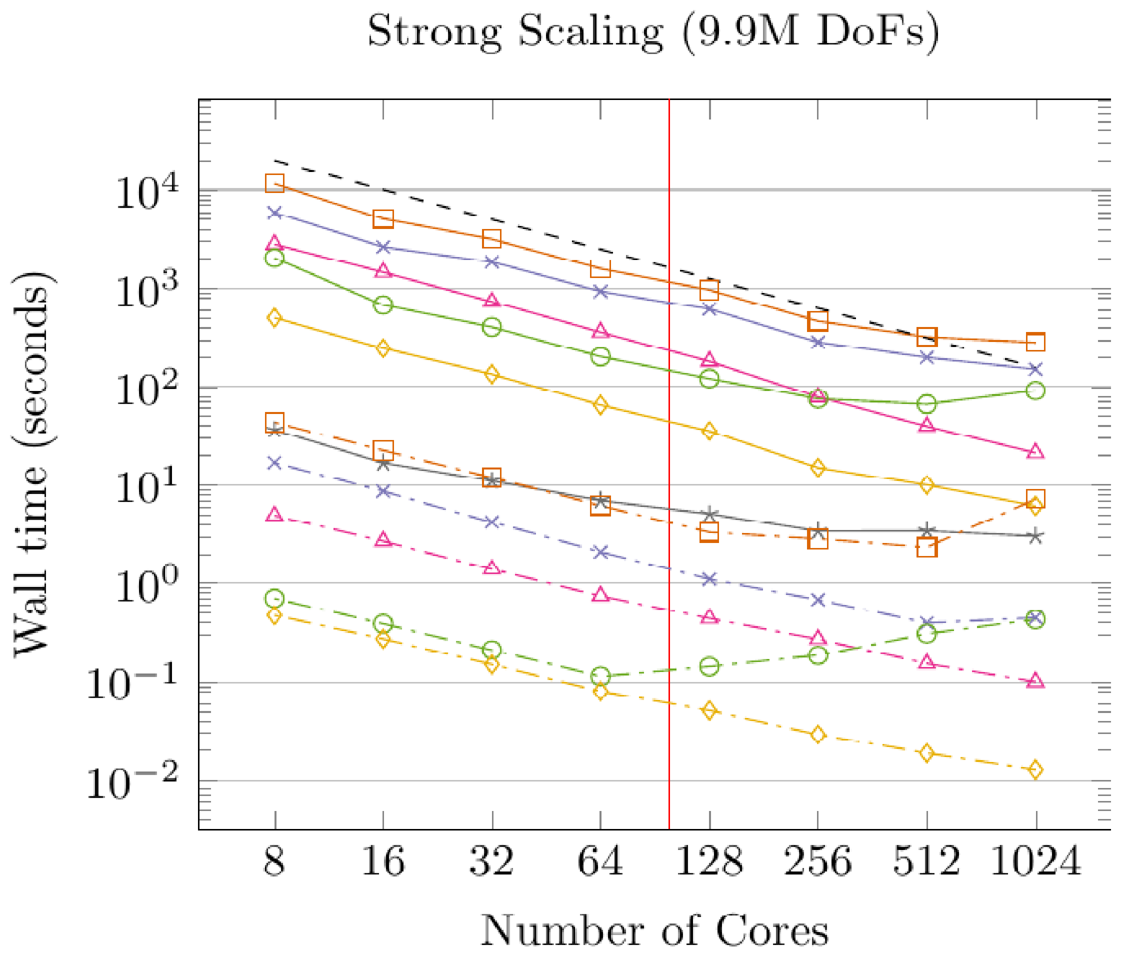
Examples (strong scaling):



Laplace equation (from Bangerth et al., 2011)

Parallel preconditioners

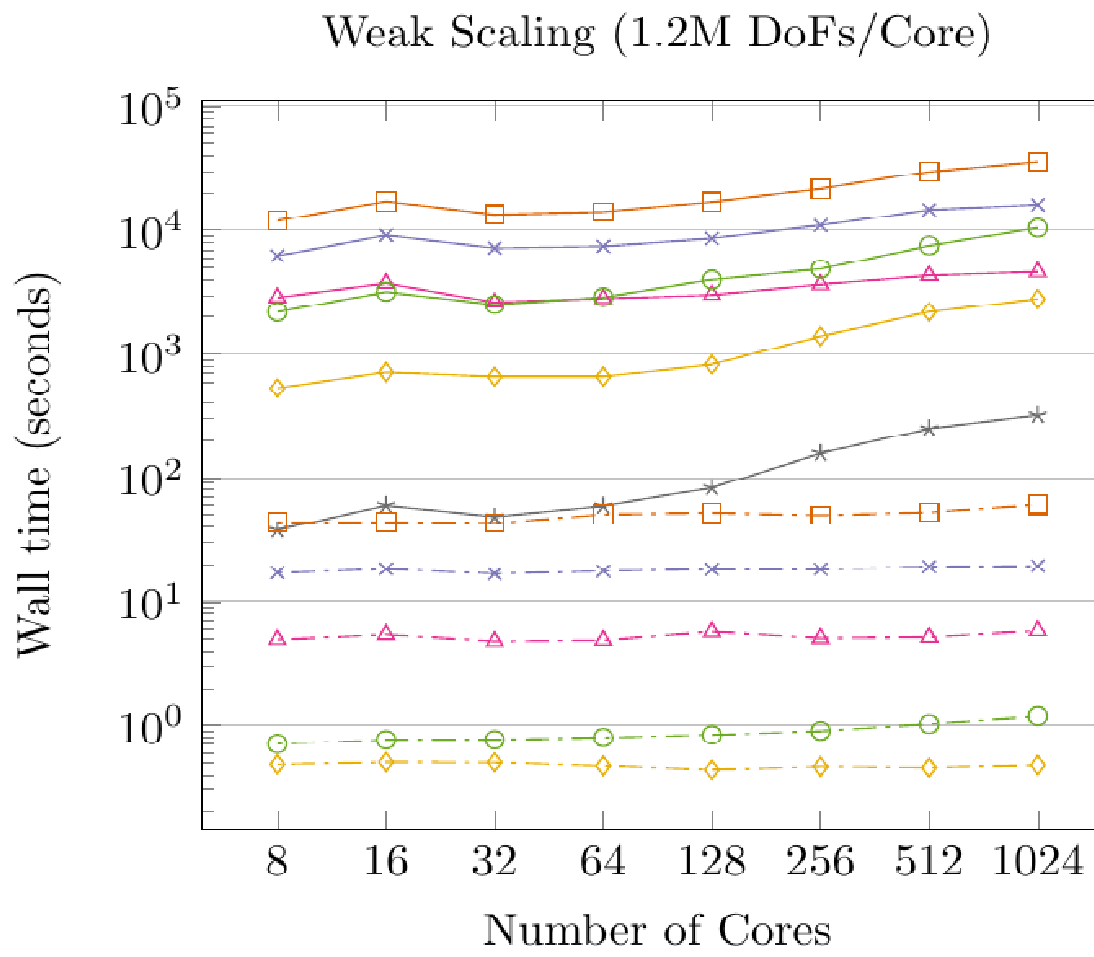
Examples (strong scaling):



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