Linear algebra libraries for high performance scientific computing
An overview of some linear algebra freeware

Stéphane Lanteri

INRIA, NACHOS project-team, 2004 Route des Lucioles, BP 93
06902 Sophia Antipolis Cedex, France
E-mail: Stephane.Lanteri@inria.fr
1. Some generalities

2. Overview of some freeware packages
   - PETSc
   - Trilinos
   - Hypre

3. A concrete example
   - The PORFLOW™ CFD software
   - Improving PORFLOW™ performances
   - Results
Some generalities

Overview of some freeware packages
- PETSc
- Trilinos
- Hypre

A concrete example
- The PORFLOW™ CFD software
- Improving PORFLOW™ performances
- Results
Some generalities

- The operations $Ax = b$ and $Ax = \lambda x$ are central numerical kernels in the solution of numerous scientific computing problems
  - Numerical solution of systems of PDEs:
    - finite difference, finite element and finite volume methods,
    - boundary element methods,
    - structured (cartesian, body-fitted) grids,
    - unstructured grids.
  - Relevant topics:
    - sparse versus dense systems,
    - symmetric versus non-symmetric systems,
    - definite versus indefinite systems,
    - direct versus iterative methods,
    - arithmetic types (real versus complex),
    - use of low level, highly optimized, numerical kernels (BLAS),
    - programming languages (Fortran 77/95, C, C++),
    - sequential versus parallel computing,
    - shared memory versus distributed memory parallel computing,
    - portability,
    - scalability.
Some generalities

- Linear system solution strategies
  - Direct methods
    - Factorization methods
    - Solution process (triangular solves)
  - Iterative methods
    - Relaxation methods (Jacobi, Gauss-Seidel, etc.)
    - Krylov methods (CG, CGS, GMRES, BiCGStab, etc.)
    - Multigrid (multilevel) methods (geometric versus algebraic)
    - Domain decomposition (multiplicative and additive Schwarz methods, Schur complement type methods)
  - Preconditioning techniques
    - Incomplete factorization (ICC, ILU(k), ILU(k,τ), BILU variants, etc.)
    - Various variants of AMG
    - Various variants of SPAI (SParse Approximate Inverse)
    - Domain decomposition (multiplicative and additive Schwarz methods, Schur complement type methods)
    - Others ...

S. Lanteri (INRIA Sophia Antipolis)
Some generalities

- Linear algebra freeware
  - Source code for a single method
  - Toolkits for a certain type of methods or/and problems
  - General packages or environments
- Programming models
  - Classical, procedural
  - Object oriented
- User data input models
  - Assembled matrix in a widely adopted format (CSR, etc.)
  - User subroutines
    - Matrix-vector product with $A$ and $A^T$
    - Preconditioning operation
Some generalities

http://www.netlib.org/utk/people/JackDongarra/la-sw.html
Some generalities

<table>
<thead>
<tr>
<th>DIRECT SOLVERS</th>
<th>DIRECT SPARSE SOLVERS</th>
<th>Real</th>
<th>Complex</th>
<th>f77</th>
<th>c</th>
<th>c++</th>
<th>Seq</th>
<th>Dist</th>
<th>SPD</th>
<th>Gen</th>
<th>SPD</th>
<th>Gen</th>
<th>Sym</th>
<th>Gen</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLAME</td>
<td>LGPL</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAPACK</td>
<td>BSD</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAPACK95</td>
<td>BSD</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>95</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NAPACK</td>
<td>BSD</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PLAPACK</td>
<td>?</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRISM</td>
<td>?</td>
<td>no</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ScalAPACK</td>
<td>BSD</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>M/P</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trilinos/Pliris</td>
<td>LGPL</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>M</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

http://www.netlib.org/utk/people/JackDongarra/la-sw.html
## Some generalities

<table>
<thead>
<tr>
<th>PRECONDITIONERS</th>
<th>Real</th>
<th>Complex</th>
<th>f77</th>
<th>c</th>
<th>c++</th>
<th>Seq</th>
<th>Dist</th>
<th>SPD</th>
<th>Gen</th>
<th>SPD</th>
<th>Gen</th>
<th>Sym</th>
<th>Gen</th>
</tr>
</thead>
<tbody>
<tr>
<td>BPKIT</td>
<td>?</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PARPRE</td>
<td>?</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPAI</td>
<td>?</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tritons/ML</td>
<td>LGPL</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### SPARSE ITERATIVE SOLVERS

<table>
<thead>
<tr>
<th></th>
<th>License</th>
<th>Support</th>
<th>Real</th>
<th>Complex</th>
<th>f77</th>
<th>c</th>
<th>c++</th>
<th>Seq</th>
<th>Dist</th>
<th>SPD</th>
<th>Gen</th>
<th>SPD</th>
<th>Gen</th>
<th>Sym</th>
<th>Gen</th>
</tr>
</thead>
<tbody>
<tr>
<td>BILUM</td>
<td>?</td>
<td>no</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BlockSolve95</td>
<td>?</td>
<td>?</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>M</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CERFACS</td>
<td>?</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>x</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GMM++</td>
<td>LGPL</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HYPRE</td>
<td>?</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>M</td>
<td></td>
<td>P</td>
<td>P</td>
<td>P</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IML++</td>
<td>?</td>
<td>?</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ISTL</td>
<td>LGPL</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>M</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ITL</td>
<td>Own</td>
<td>yes</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ITPACK</td>
<td>PD</td>
<td>?</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LASPack</td>
<td>?</td>
<td>?</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSQR</td>
<td>CPL</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

http://www.netlib.org/utk/people/JackDongarra/la-sw.html
Some generalities

<table>
<thead>
<tr>
<th>Library</th>
<th>Ownership</th>
<th>Developer</th>
<th>M</th>
<th>X</th>
<th>X</th>
<th>X</th>
<th>M</th>
<th>X</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>pARMS</td>
<td>?</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>M</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>PETSc</td>
<td>Own</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>PIM</td>
<td>?</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>PSBLAS</td>
<td>Own</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>M</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>P-SparsLIB</td>
<td>?</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>M</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>OMRPACK</td>
<td>?</td>
<td>?</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>SLAP</td>
<td>PD</td>
<td>?</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>SPLIB</td>
<td>?</td>
<td>?</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>SYMMLQ</td>
<td>CPL</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Trilinos</td>
<td>LGPL</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>M</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Templates</td>
<td>BSD</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

http://www.netlib.org/utk/people/JackDongarra/la-sw.html
Some generalities

<table>
<thead>
<tr>
<th>SPARSE EIGENVALUE SOLVERS</th>
<th>License</th>
<th>Support</th>
<th>Real</th>
<th>Complex</th>
<th>f77</th>
<th>c</th>
<th>c++</th>
<th>Seq</th>
<th>Dist</th>
<th>SPD</th>
<th>Gen</th>
<th>SPD</th>
<th>Gen</th>
<th>Sym</th>
<th>Gen</th>
</tr>
</thead>
<tbody>
<tr>
<td>LZPACK</td>
<td>?</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>M/P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LASO</td>
<td>PD</td>
<td>?</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PARPACK</td>
<td>?</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>M/P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>PRIMME</td>
<td>LGPL</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>M/P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SLEPc</td>
<td>Own</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPAM</td>
<td>?</td>
<td>yes</td>
<td>X</td>
<td>90</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trilinos/Anasazi</td>
<td>LGPL</td>
<td>yes</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRLAN</td>
<td>Own</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

http://www.netlib.org/utk/people/JackDongarra/la-sw.html
1 Some generalities

2 Overview of some freeware packages
   - PETSc
   - Trilinos
   - Hypre

3 A concrete example
   - The PORFLOW™ CFD software
   - Improving PORFLOW™ performances
   - Results
Some generalities

Overview of some freeware packages
- PETSc
- Trilinos
- Hypre

A concrete example
- The PORFLOW™ CFD software
- Improving PORFLOW™ performances
- Results
The PETSc toolkit

A toolkit that can ease the development of parallel, non-trivial PDE solvers (and reduce development time)

What is PETSc?

- A freely available and supported research code
  - Developed by a team from the Mathematics and Computer Science department at Argonne National Laboratory
  - Begun in September 1991
  - Current version: **2.3.3** (May 2007)
  - [http://www.mcs.anl.gov/petsc](http://www.mcs.anl.gov/petsc)
  - Hyperlinked documentation and manual pages for all routines
  - Usable from Fortran 77/90, C and C++

- Portable to any parallel system supporting MPI
The **PETSc** toolkit

- **PETSc** consists of a variety of libraries
- Each library manipulates a particular family of objects and the operations one would like to perform on the objects
- Some of the **PETSc** modules deal with:
  - index sets, including permutations, for indexing into vectors, renumbering, etc.
  - vectors,
  - matrices (generally sparse),
  - distributed arrays (useful for parallelizing regular grid-based problems),
  - Krylov subspace methods,
  - preconditioners including multigrid methods,
  - sparse direct solvers,
  - nonlinear solvers,
  - timesteppers for solving time-dependent (nonlinear) PDEs.
- Each library consists of an abstract interface (simply a set of calling sequences) and one or more implementations using particular data structures
The **PETSc** toolkit

- **Interfaced (external) solvers**
  - LU (sequential)
    - SuperLU (LBNL)
    - ESSL (IBM)
    - Matlab
    - LUSOL (U. Stanford)
  - Parallel LU
    - SuperLU (LBNL)
    - SPOOLES (Boeing)
    - MUMPS (Cerfacs)
  - Parallel Cholesky
    - DSCPACK (Penn State U.)
  - Sparse Approximate Inverse
    - ParaSails (part of **HYPRE**, LLNL)
    - SPAI (S. Barnard - NASA Ames and M. Grote - ETH Zurich)
The PETSc toolkit

- Interfaced (external) solvers
  - Algebraic Multigrid
    - BoomerAMG (parallel, part of HYPRE, LLNL)
    - RAMG (sequential, J. Ruge and K. Stueben’s original code)
    - SAMG (sequential, K. Stueben’s version for systems of PDEs)
  - ILU (sequential)
    - ILUT (Y. Saad’s code included in SPARSKIT, U. Minesota)
  - Parallel ILU
    - BlockSolve95 (ANL)
    - PILUT (part of HYPRE, LLNL)
    - Euclid (Old Dominion U./LLNL)
  - Parallel ICC
    - BlockSolve95 (ANL)
The PETSc toolkit

- Structure of PETSc

![Diagram of PETSc structure](image)
The PETSc toolkit

- Flow control of a PDE solver
Levels of abstraction in mathematical software

1. Application-specific interface
   - Programmer manipulates objects associated with the application

2. High-level mathematics interface
   - Programmer manipulates mathematical objects (weak forms, boundary conditions meshes, etc.)

3. Algorithmic and discrete mathematics interface
   - Programmer manipulates mathematical objects (vectors, matrices, linear and non-linear systems of equations, etc.)
   - Programmer manipulates algorithmic objects (linear and non-linear solvers, preconditioners, etc.)

4. Low-level computational kernels
   - BLAS-type operations, FFT, etc.
The PETSc toolkit

- Object-oriented programming and design
  - Design is based not on the data in object but instead on operations performed with or on the data
  - For example, a vector is not a 1D array of numbers but an abstract object where addition and scalar multiplication are defined
  - Added difficulty is the efficient use of computer

- The PETSc programming model
  - Goals
    - Portability, efficiency and scalability
  - Approach
    - Distributed memory model
    - Still exploits compiler discovered parallelism on each node (SMP)
    - Hide within objects the details of communication operations
    - User orchestrates communications at a higher abstract level
The **PETSc** toolkit

- **What is not in PETSc**
  - High level representation of PDEs
  - Load balancing
  - Sophisticated visualization capabilities

  **But PETSc does interface to external software that provides some of this functionality**

- **PETSc** is a set of library interfaces
  - The same interfaces are provided in C, C++, Fortran 77 and Fortran 90

- **Profiling**
  - Event timing
  - Memory usage
  - MPI messages
The PETSc toolkit

- **Linear algebra: vectors**
  - **What are PETSc vectors?**
    - One of the simplest PETSc object: Vec
    - Used for storing field solutions, right-hand sides, etc.
    - Each process locally owns a subvector of contiguously numbered global indices
  - **Basic functions (collective operations)**
    - **VecCreate(MPI_Comm comm, Vec *v)**
      - comm is the MPI communicator for the processes that share the vector
      - Automatically generates the appropriate vector type, sequential or parallel over all processes in comm
    - **VecSetSizes(Vec v, int M, int m)**
      - M is the total number of vector components
      - m is the number of vector components to store on local process
    - **VecSetFromOptions(Vec v)**
      - Defines vector characteristics from command line arguments
    - **VecSetType(Vec v, VecType)**
      - VecType is VEC_SEQ or VEC_MPI or VEC_SHARED
Linear algebra: vectors

Assigning values to individual components of a vector

A three-step process

1. Each process tells PETSc what values to set or add to a vector component
2. Begin communications to ensure that values end up where needed
3. Complete the communications

Other operations can proceed between 2. and 3.

Basic functions

- `VecSetValues(Vec v, int n, int *indices, PetscScalar *values, INSERT_VALUES)`
  - `n` is the number of elements to insert
  - `indices` contains the global component indices
  - `values` is the array of values to be inserted

- `VecAssemblyBegin(Vec v)`
- `VecAssemblyEnd(Vec v)`
- `VecSetValues` can be called with `ADD_VALUES`

Releasing a vector: `VecDestroy(Vec v)`
The PETSc toolkit

Linear algebra: vectors

- One way to set the elements of a vector
  
  ```c
  VecGetSize(x, &N); /* Global size of the vector */
  MPI_Comm_rank(PETSC_COMM_WORLD, &rank)
  if (rank == 0) {
      for (i=0; i<N; i++)
          VecSetValues(x, 1, &i, &i, INSERT_VALUES);
  }
  /* These two routines ensure that the data is distributed to other processes */
  VecAssemblyBegin(x);
  VecAssemblyEnd(x);
  
  - A parallel way to set the elements of a vector
    ```c
    VecGetOwnershipRange(x, &low, &high);
    for (i=low; i<high; i++)
        VecSetValues(x, 1, &i, &i, INSERT_VALUES);
    /* These two routines must be called (in case some other processes contributed a value owned by another process */
    VecAssemblyBegin(x);
    VecAssemblyEnd(x);
    ```
Linear algebra: selected vector operations

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>VecAXPY(Scalar *a, Vec x, Vec y)</td>
<td>y = y + a*x</td>
</tr>
<tr>
<td>VecAYPX(Scalar *a, Vec x, Vec y)</td>
<td>y = x + a*y</td>
</tr>
<tr>
<td>VecWAXPY(Scalar *a, Vec x, Vec y, Vec w)</td>
<td>w = a*x + y</td>
</tr>
<tr>
<td>VecScale(Scalar *a, Vec x)</td>
<td>x = a*x</td>
</tr>
<tr>
<td>VecCopy(Vec x, Vec y)</td>
<td>y = x</td>
</tr>
<tr>
<td>VecPointwiseMult(Vec x, Vec y, Vec w)</td>
<td>w_i = x_i * y_i</td>
</tr>
<tr>
<td>VecMax(Vec x, int *idx, double *r)</td>
<td>r = max x_i</td>
</tr>
<tr>
<td>VecShift(Scalar *s, Vec x)</td>
<td>x_i = s + x_i</td>
</tr>
<tr>
<td>VecAbs(Vec x)</td>
<td>x_i =</td>
</tr>
<tr>
<td>VecNorm(Vec x, Norm Type type, double *r)</td>
<td>r =</td>
</tr>
</tbody>
</table>
The PETSc toolkit

- Linear algebra: matrices
  - Must use `MatSetValues()`
    - Automatic communication during assembly
  - Supports many storage formats
    - IJ, block IJ, symmetric IJ, block diagonal, etc.
  - Supports data structures for several packages
    - Spooles, MUMPS, SuperLU, UMFPack, DSCPack
- Matrices and polymorphism
  - Single user interface for matrix operations
    - Matrix assembly (`MatSetValues()`)
    - Matrix/vector multiplication (`MatMult()`)
    - Matrix viewing (`MatView()`)
  - Multiple underlying implementations
- A matrix is defined by its interfaces (the operations that you can perform with it), not by its data structure
Linear algebra: matrices

Matrix assembly: `MatSetValues(Mat, ...)`
- Number of rows to insert or add
- Number of columns to insert or add
- Global indices of rows and columns
- Values to insert or add
- Mode: `INSERT_VALUES` or `ADD_VALUES`

Completion: `MatAssemblyBegin(Mat)` and `MatAssemblyEnd(Mat)`

Parallel matrix distribution
- Each process locally owns a submatrix of contiguously numbered global rows
- `MatGetOwnershipRange(Mat A, int *rstart, int *rend)`
  - `rstart`: first locally owned row of the global matrix
  - `rend-1`: last locally owned row of the global matrix
Parallel data layout: ghost values

To evaluate a function $f(x)$, each process requires its local portion of the vector $x$ as well as its ghost values or bordering portions of $x$ that are owned by neighboring processes.
Parallel data layout: communication and physical discretization
The PETSc toolkit

- Parallel data layout: structured grids
  - Global representation: each process stores a unique local set of vertices (each vertex is owned by exactly one process)
  - Local representation: each process stores a unique local set of vertices as well as ghost nodes from neighboring processes
Parallel data layout: Distributed Arrays (DA)

- They are used in conjunction with PETSc vectors
- They are intended for use with logically regular rectangular grids
- They are designed to work with data stored in standard multidimensional arrays

⇒ DAs are not intended for parallelizing unstructured grid problems

The PETSc DA object manages the parallel communication operations required while working with data stored in regular arrays

- The actual data is stored in appropriately sized vector objects
- The DA object only contains the parallel data layout information and communication information
- The DA object may be used to create vectors and matrices with the proper layout
**PETSc toolkit**

- Parallel data layout: Distributed Arrays (DA)
  - Basic functions
    - Create a 2D DA
      
      ```c
      DACreate2d(MPI_Comm comm, DAPeriodicType wrap, DASTencilType st,
                  int M, int N, int m, int n, int dof, int s,
                  int *lx, int *ly, DA *da)
      ```

      - M and N are the global numbers of grid points in each direction
      - m and n define the process partition in each direction
        (with m*n = # processes in the communicator `comm`)
      - `wrap` can be DA_NONPERIODIC, DA_XPERIODIC, DA_YPERIODIC
        or DA_XYPERIODIC
      - `dof` is the number of degrees of freedom at each array point
      - `s` is the stencil width (i.e the width of the ghost point region)
      - `lx` and `ly` are arrays that may contain the number of nodes
        along the x and y axis for each process (i.e the dimensions of `lx` and `ly` are `m` and `n`); can be set to PETSC_NULL
      - `st` can be DA_STENCIL_STAR or DA_STENCIL_BOX

- Similar functions for 1D and 3D arrays: DACreate1d and DACreate3d
Parallel data layout: Distributed Arrays (DA)

- Basic functions
  - Each DA object defines the layout of two vectors: a distributed global vector and a local vector that includes room for the appropriate ghost points.
  - The DA object does not internally allocate any associated storage space for field values.
  - The user creates vector objects that use the DA layout information:
    - DACreateGlobalVector(DA da, Vec *g)
    - DACreateLocalVector(DA da, Vec *l)
  - Several vectors can share the same DA (i.e., share the same communication information).
Parallel data layout: unstructured grids

To facilitate general scatter and gather operations on vectors, PETSc employs the concept of an Index Set (IS)

Creation of an IS based on a list of integers

```
ISCreateGeneral(MPI_Comm comm, int n, int *indices, IS *is)
```

Copy selected components from one vector to another

```
VecScatterCreate(Vec x, IS ix, Vec y, IS iy, VecScatter *ctx)
VecScatterBegin(Vec x, Vec y, INSERT_VALUES,
              SCATTER_FORWARD, VecScatter ctx)
VecScatterEnd(Vec x, Vec y, INSERT_VALUES,
              SCATTER_FORWARD, VecScatter ctx)
VecScatterDestroy(VecScatter ctx)
```

The vectors can be sequential or parallel

The IS ix and iy must have the same number of entries

ADD_VALUES can be used instead of INSERT_VALUES
The PETSc toolkit

- PETSc linear solvers: the KSP object
  - Provides uniform and efficient access to PETSc linear solvers
  - Sequential and parallel, direct and iterative methods
  - KSP uses the same calling sequence for both direct and iterative solution of a linear system
  - Basic functions
    - **KSPCreate(MPI_Comm comm, KSP *ksp)**
    - **KSPSetOperators(KSP ksp, Mat amat, Mat pmat, MatStructure flag)**
      - Sets the matrices associated with the linear system
      - **Amat** is a symbolic place holder for any kind of matrix (PETSc supports matrix-free methods)
      - **Pmat** is the matrix from which the preconditioner is to be constructed
      - **flag** can be used to eliminate unnecessary work when repeatedly solving linear systems of the same size with the same preconditioning method
    - **flag** can be **SAME_NONZERO_PATTERN**, **DIFFERENT_NONZERO_PATTERN** or **SAME_PRECONDITIONER**
The PETSc toolkit

- **PETSc linear solvers: the KSP object**
  - Basic functions
    - `KSPSolve(KSP ksp, Vec b, Vec x)`
    - `KSPGetIterationNumber(KSP ksp, int *its)`
    - `KSPDestroy(KSP ksp)`
  - KSP can be configured from command line options
    - `KSPSetFromOptions(KSP ksp)`
  - Default solver: GMRES(m)
  - Default preconditioner:
    - sequential case: ILU(0),
    - parallel case: BJ (Block Jacobi).
  - Krylov methods: `KSPSetType(KSP ksp, KSPType method)`
    - `method` can be `KSPRICHARDSON`, `KSPCHEBYCHEV`, `KSPCG`, `KSPGMRES`, `KSTFQMR`, `KSPBCGS`, `KSPBICG`, etc.
    - Various functions exist for method-specific options
PETSc linear solvers: the KSP object

- Preconditioning: $[M_L^{-1} A M_R^{-1}](M_R x) = M_L^{-1} b$
  - Left preconditioning is used by default
  - Right preconditioning available for some methods
    \[
    \text{KSPSetPreconditionerSide(KSP ksp, PCSide PC\_RIGHT)}
    \]

- Convergence is detected if $\| r_k \|_2 < \max(\text{rtol} \times \| b \|_2, \text{atol})$
  - $r_k$ is the true or the preconditioned residual depending on the solution method
  - rtol is the relative decrease in the residual
  - atol is the absolute size of the residual

- Divergence is detected if $\| r_k \|_2 > \text{dtol} \times \| b \|_2$
  - dtol is the relative increase in the residual

\[
\text{KSPSetTolerances(KSP ksp, double rtol, double atol, double dtol, int maxits)}
\]

- Since the convergence of Krylov subspace methods depends strongly on the spectrum of the preconditioned operator, PETSc provides routines for eigenvalue approximation via the Arnoldi or Lanczos iteration
The PETSc toolkit

- **PETSc** linear solvers: the **KSP** object
- Preconditioners: \texttt{PCSetType(PC pc, PCType method)}

<table>
<thead>
<tr>
<th>Method</th>
<th>PCType</th>
<th>Option name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacobi</td>
<td>PCJACOBI</td>
<td>jaciobi</td>
</tr>
<tr>
<td>Block Jacobi</td>
<td>PCBJACOBI</td>
<td>bjacobi</td>
</tr>
<tr>
<td>SOR (and SSOR)</td>
<td>PCSOR</td>
<td>sor</td>
</tr>
<tr>
<td>Incomplete Cholesky</td>
<td>PCICCC</td>
<td>icc</td>
</tr>
<tr>
<td>Incomplete LU</td>
<td>PCILU</td>
<td>ilu</td>
</tr>
<tr>
<td>Additive Schwarz</td>
<td>PCASMD</td>
<td>asm</td>
</tr>
<tr>
<td>Linear solver</td>
<td>PCKSPB</td>
<td>ksp</td>
</tr>
<tr>
<td>Combination of preconditioners</td>
<td>PCCOMPOSITE</td>
<td>composite</td>
</tr>
<tr>
<td>LU</td>
<td>PCLU</td>
<td>lu</td>
</tr>
<tr>
<td>Cholesky</td>
<td>PCCholesky</td>
<td>cholesky</td>
</tr>
<tr>
<td>No preconditioner</td>
<td>PCNONE</td>
<td>none</td>
</tr>
<tr>
<td>Shell for user-defined PC</td>
<td>PCSHELL</td>
<td>shell</td>
</tr>
</tbody>
</table>
The PETSc toolkit

- **PETSc preconditioners**
  - ILU and ICC
    - `PCILUSetLevels(PC pc, int levels)`
    - `PCICCSSetLevels(PC pc, int levels)`
    - `PCILUSetUseDropTolerance(PC pc, double dt, double dtcol, int dtcount)`
    - `PCILUDTSetReuseOrdering(PC pc, PetscTruth flag)`
    - `PCILUDTSetReuseFill(PC pc, PetscTruth flag)`
      - Multiple solves with the same KSP context: reuse the ordering (or the fill) computed during the first (drop tolerance) factorization
    - `PCILUSetUseInPlace(PC pc)`
    - `PCILUSetAllowDiagonalFill(PC pc)`

S. Lanteri (INRIA Sophia Antipolis)
PETSc toolkit

- **PETSc preconditioners**
  - **SOR and SSOR**
    - `PCSORSetOmega(PC pc, double omega)`
    - `PCSORSetIterations(PC pc, int its, int lits)`
      - `its` is the number of inner iterations
      - `lits` is the number of local iterations (i.e. the number of smoothing sweeps on a process before doing a ghost point update from the other processes)
      - The total number of SOR sweeps is `its x lits`
    - `PCSORSetSymmetric(PC pc, MatSORType type)`
      - `type` can be `SOR_FORWARD_SWEEP`, `SOR_BACKWARD_SWEEP` or `SOR_SYMMETRIC_SWEEP` (i.e. SSOR)
The PETSc toolkit

- **PETSc** preconditioners
  - Block Jacobi and overlapping Additive Schwarz
    - Parallel preconditioners (Block Gauss-Seidel in sequential)
    - By default, ILU(0) factorization on each individual block
    - `PCBJacobiGetSubKSP(PC pc, int *n_local, int *first_local, KSP **subksp)`
    - `PCASMGGetSubKSP(PC pc, int *n_local, int *first_local, KSP **subksp)`
      - Extract the KSP context for each local block
      - `n_local` indicates the number of blocks on the calling process
      - `first_local` indicates the global number of the first block on the calling process (blocks are numbered successively by the processes from 0 to gb - 1 where gb is the global number blocks)
      - `subksp` is the array of KSP contexts for the local blocks
    - Through the array of KSP contexts, the user is allowed to set different solvers for the various blocks
PETSc preconditioners

- Block Jacobi and overlapping Additive Schwarz
  - PCBJacobiSetTotalBlocks(PC pc, int blocks, int *size)
    - size is an array indicating the size of each block
  - PCASMSSetTotalSubdomains(PC pc, int n, IS *is)
  - Also: PCBJacobiSetLocalBlocks and PCASMSSetLocalSubdomains
  - PCASMSSetType(PC pc, PCASMTYPE type)
    - type defines the type of ASM method
    - PC_ASM_BASIC: standard method (full restriction and interpolation)
    - PC_ASM_RESTRICT: full restriction and ignore off-process values during interpolation
    - PC_ASM_INTERPOLATE: full interpolation and limited restriction
    - PC_ASM_NONE: limited restriction and interpolation
  - PCASMSSetOverlap(PC pc, int overlap)
    - Set the overlap to compute in constructing the subdomains
    - Default size: 1
The PETSc toolkit

- **PETSc preconditioners**
  - **Multigrid**
    - In the PC framework, the user is required to provide the coarse grid solver, smoothers, restriction and interpolation as well as the code to compute residuals
    - Construction of a multigrid preconditioner
      - `KSPCreate(MPI_Comm comm, KSP *ksp)`
      - `KSPGetPC(KSP ksp, PC *pc)`
      - `PCSetType(PC pc, PCMG)`
      - `MGSetLevels(PC pc, int levels, MPI_Comm *comms)`
    - Multigrid algorithm: `MGSetType(PC pc, MGType mode)`
      - Standard V or W-cycle: `MGMULTIPLICATIVE`
      - Additive form: `MGADDITIVE`
      - Full multigrid: `MGFULL`
      - Kaskade algorithm: `MGKASKADE`
    - Multigrid cycle: `MGSetCycles(PC pc, int cycles)`
    - Smoothing steps: `MGSetNumberSmoothUp(PC pc, int m)` and `MGSetNumberSmoothDown(PC pc, int n)`
The PETSc toolkit

- **SNES** (Scalable Nonlinear Equations Solvers)
  - **Goal**: for problems arising from PDEs, supports the general solution of $F(x) = 0$
  - Newton-based methods, including:
    - line search strategies,
    - trust region approaches,
    - pseudo-transient continuation,
    - matrix-free variants.

- Create of a SNES solver
  
  `SNESCreate(MPI_Comm comm, SNES *snes)`

- Select of a nonlinear solution method
  
  `SNESSetType(SNES snes, SNESType method)`

- Solution process
  
  `SNESSolve(SNES snes, Vec x)`

- Release a SNES context
  
  `SNESDestroy(SNES snes)`
SNES (Scalable Nonlinear Equations Solvers)

- Solvers based on callbacks
  - User provides routines to perform actions that the library requires
  - Example: `SNESetFunction(SNES snes, ...)`

with parameters:

- `UserVector`: vector to store function values
- `UserFunction`: name of the user’s function
- `UserContext`: pointer to private data for the user’s function

Whenever the library needs to evaluate the user’s nonlinear function, the solver may call the application code directly with its own local state.

**UserContext** serves as an application context object (data are handled through such opaque objects and the library never sees irrelevant application data.)
The **PETSc** toolkit

- **SNES** (Scalable Nonlinear Equations Solvers)
  - Nonlinear function evaluation

  ```c
  SNESetFunction(SNES snes, Vec f,
                 PetscErrorCode(*FormFunction)(SNES snes, Vec x, Vec f, void *ctx),
                 void *ctx)
  ```

  - The vector `f` stores the evaluation of the function at the vector `x` i.e. $F(x)$
  - `ctx` is an optional user-defined context used to store application-specific data

- Jacobian evaluation: user provides code to evaluate the Jacobian of $F(x)$ or he can make use of appropriate **PETSc** functionalities (sparse finite difference approximation or automatic differentiation)

  ```c
  SNESetJacobian(SNES snes, Mat A, Mat B
                 PetscErrorCode(*FormJacobian)(SNES snes, Vec x, Mat *A, Mat *B, MatStructure *flag,
                                              void *ctx), void *ctx)
  ```

  - `A` is the Jacobian matrix and `B` is the preconditioner
  - `flag` indicates information about the preconditioner matrix structure
**SNES (Scalable Nonlinear Equations Solvers)**

**Sample application context**

```c
typedef struct{
    /* Basic application data */
    double velocity, pressure; /* Physical parameters */
    int mx, my; /* Discretization parameters */
    int mc; /* Number of d.o.f per node */

    /* Parallel data */
    MPI_Comm comm; /* MPI communicator */
    DA da; /* Distributed array */
    Vec localF, localX; /* Local ghost vectors */
} AppCtx;
```
Using PETSc with other packages

- **HYPRE** (preconditioners from LLNL/CASC)
- **TAO** (Toolkit for Advanced Optimization from ANL/MCS)
- **PVODE** (ODE integrators from LLNL/CASC)
- **MATLAB** from Mathworks
- **PARMETIS** (parallel graph partitioning from U. Minnesota)
Parallel solution of a linear system with SLES
The PETSc toolkit: a sample code

- Related links
  - Web page of PETSc
    http://acts.nersc.gov/petsc/
  - The Advanced CompuTational Software (ACTS) collection project
    http://acts.nersc.gov/project.html
1 Some generalities

2 Overview of some freeware packages
   - PETSc
   - Trilinos
   - Hypre

3 A concrete example
   - The PORFLOW™ CFD software
   - Improving PORFLOW™ performances
   - Results
The **Trilinos** project

- Coordinated effort between several teams at Sandia National Laboratory (Albuquerque, NM)
  - Computational Mathematics and Algorithms Department
  - Computational Sciences and Mathematics Research Department
  - Computational Sciences Department
  - High Performance and Networking Department
- Grew out of a group of established numerical algorithms efforts at SNL (e.g. the Aztec library of preconditioned Krylov linear solvers)
- LGPL (Gnu Lesser General Public License)
- Current version: 8.0.5 (January 2008)
- **Goal**: to develop algorithms and enabling technologies within an object-oriented software framework for the solution of large-scale, complex multi-physics engineering and scientific problems
Main features of the Trilinos design philosophy

- Object oriented framework
- Abstract interfaces for flexible interoperability of components
- Full featured set of concrete classes
- Two-level software structure designed around collections of packages
- A Trilinos package is an integral unit developed by a small team of experts in a particular algorithms area
- The Trilinos design allows individual packages to grow and mature autonomously to the extent the algorithms and package developers dictate
- Packages exist underneath the Trilinos top level
- Trilinos top level provides a common look-and-feel, including configuration, documentation, licensing and bug-tracking
The Trilinos project

- Primarily written in C++
- Some C and Fortran user support
- Provides an open architecture that allows easy integration of external software packages

Services provided by Trilinos
- Configuration management
  - Based on Autoconf and Automake
- Regression testing
- Automatic testing
- Portable interface to BLAS and LAPACK
  - Call to BLAS and LAPACK functions from C++ (and C)
- Source code repository (CVS)
- Quick-start package infrastructure
  - Provides a starting point for project organization, Autotools usage, automatic generation of documentation (Doxygen), regression testing and Web site set up
The **Trilinos** project

- **Trilinos** packages
  - A self-contained, independent piece of software
  - With its own set of requirements, its own development team and group of users
  - **Trilinos** itself is designed to respect the autonomy of packages
  - **Trilinos** offers a variety of ways for a particular package to interact with other **Trilinos** packages
  - Also offers a set of tools that can assist package developers with builds across multiple platforms, generating documentation and regression testing across a set of target platforms
  - What a package must do to be called a **Trilinos** package is minimal, and varies with each package
The **Trilinos** project

- **Trilinos** packages: basic linear algebra libraries
  - **Epetra**
    - Core linear algebra package
    - Facilitates construction and manipulation of distributed and serial graphs, sparse and dense matrices, vectors and multivectors
  - **EpetraExt**
    - Extensions to the core linear algebra package, Epetra
  - **Tpetra**
    - Next-generation, templated version of Petra, taking advantage of the newer advanced features of C++
  - **Jpetra**
    - Experimental Java version of the Petra library
  - **Kokkos**
    - Core kernel package
The **Trilinos** project

**Trilinos packages: preconditioners**

- **AztecOO**
  - ILU-type preconditioner

- **IFPACK**
  - Distributed algebraic preconditioner package
  - Includes incomplete factorizations and relaxation-based preconditioners in domain decomposition framework
  - Compatible with AztecOO

- **ML**
  - Multilevel, distributed memory algebraic preconditioners
  - Provides multi-level, multigrid-like preconditioners for distributed linear systems
  - Compatible with AztecOO

- **Meros**
  - Segregated preconditioning package
  - Provides scalable block preconditioning for problems that coupled simultaneous solution variables such as Navier-Stokes problems
The Trilinos project

Trilinos packages: linear solvers

- Epetra
  - Provides wrappers for select BLAS and LAPACK routines
- Teuchos
  - Provides wrappers for select BLAS and LAPACK routines
- Pliris
  - Object-oriented interface to a LU solver for dense matrices on parallel platforms
- AztecOO
  - Solves linear systems of equations via preconditioned Krylov methods
  - Uses Epetra objects
  - Compatible with IFPACK, ML and Aztec
- Belos
  - Next-generation iterative solvers written using a traits interface, meaning that it has no explicit dependence on any concrete linear algebra library
  - Can be used with any concrete linear algebra library that implements the Thyra abstract interfaces and even Epetra directly
The **Trilinos** project

- **Trilinos packages: linear solvers**
  - Amesos
    - Direct solver classes
    - Supports use of a growing list of third party direct solvers, including DSCPACK, SuperLU, SuperLUDist and UMFPACK
    - Compatible with Epetra
  - Komplex
    - Solves complex-valued linear systems via equivalent real formulations
The **Trilinos** project

- **Trilinos** packages: non-linear solvers
  - NOX
    - Abstract and concrete classes for construction and solution of nonlinear problems
  - LOCA
    - Software library for performing bifurcation analysis of large-scale applications
    - When implemented with an application code, LOCA enables the tracking of solution branches as a function of system parameters and the direct tracking of bifurcation points
    - Designed to drive application codes that use Newton’s method to locate steady-state solutions to nonlinear problems
  - MOOCHO (Multifunctional Object-Oriented arCHitecture for Optimization)
    - Designed to solve large-scale, equality and inequality nonlinearly constrained, non-convex optimization problems using reduced-space successive quadratic programming (SQP) methods
  - Rythmos
    - Transient integrator for ordinary differential equations and differential-algebraic equations with support for explicit, implicit, one-step and multi-step algorithms
    - The fundamental design of Rythmos is aimed at supporting operator-split algorithms, multi-physics applications, block linear algebra, and adjoint integration
The **Trilinos** project

- **Trilinos packages: eigensolvers**
  - Anasazi
    - Extensible and interoperable framework for solving large-scale eigenvalue algorithms
    - Provides a generic interface to a collection of algorithms

- **Trilinos packages: automatic differentiation**
  - Sacado
    - Package for automatic differentiation of C++ programs
    - Provides simple yet fast and efficient classes for forward, reverse and Taylor polynomial mode automatic differentiation using C++ template and operator overloading
    - The resulting derivatives can be leveraged in numerous ways including nonlinear solves with NOX continuation and bifurcation analysis with LOCA optimization with MOOCHO, and time integration with Rythmos
The Trilinos project

- **Trilinos packages: eigensolvers**
  - Anasazi
    - Extensible and interoperable framework for solving large-scale eigenvalue algorithms
    - Provides a generic interface to a collection of algorithms

- **Trilinos packages: automatic differentiation**
  - Sacado
    - Package for automatic differentiation of C++ programs
    - Provides simple yet fast and efficient classes for forward, reverse and Taylor polynomial mode automatic differentiation using C++ template and operator overloading
    - The resulting derivatives can be leverage in numerous ways including nonlinear solves with NOX continuation and bifurcation analysis with LOCA optimization with MOOCHO, and time integration with Rythmos
**The Trilinos project**

- **Trilinos packages: domain decomposition**
  - Claps
    - A collection of domain decomposition preconditioners and solvers
  - Moertel
    - Mortar methods that can be used in a large class of nonconforming situations such as the surface coupling of different physical models, discretization schemes or non-matching triangulations along interior interfaces of a domain

- **Trilinos packages: repartitioning/rebalancing**
  - Isorropia
    - Repartitioning/rebalancing package, intended to assist with redistributing objects such as matrices and matrix-graphs in a parallel execution setting, to allow for more efficient computations
    - Is primarily an interface to the Zoltan library, but can be built and used with minimal capability without Zoltan
The Trilinos project

- **Trilinos packages:** domain decomposition
  - Claps
    - A collection of domain decomposition preconditioners and solvers
  - Moertel
    - Mortar methods that can be used in a large class of nonconforming situations such as the surface coupling of different physical models, discretization schemes or non-matching triangulations along interior interfaces of a domain

- **Trilinos packages:** repartitioning/rebalancing
  - Isorropia
    - Repartitioning/rebalancing package, intended to assist with redistributing objects such as matrices and matrix-graphs in a parallel execution setting, to allow for more efficient computations
    - Is primarily an interface to the Zoltan library, but can be built and used with minimal capability without Zoltan
The **Trilinos** project

- **Trilinos packages**: abstract interfaces and adapters
  - Thyra
    - Abstract linear solver package
  - PyTrilinos
    - Python interfaces to selected Trilinos packages
  - WebTrilinos
    - Web interface to experiment with Trilinos through a browser
  - Stratimikos
    - Contains a unified set of Thyra-based wrappers to linear solver and preconditioner capabilities in Trilinos
    - Also a place where unified testing of linear solvers and preconditioners can be performed
The **Trilinos** project

- **Trilinos packages:** utilities
  - Teuchos
    - Common tools package
  - TriUtils
    -Utilites used by many of the **Trilinos** packages
  - EpetraExt
    - Matrix/Vector read/write utilities
  - RTOp (reduction/transformation operators)
    - Provides the basic mechanism for implementing vector operations in a flexible and efficient manner
  - Galeri
    - For generating linear systems used by many of the **Trilinos** packages for examples and tests
The **Trilinos** project

- **The Petra and TSF packages**
  - TSF provides common abstract application programmer interfaces (APIs) for other Trilinos packages
    - A collection of abstract classes that defines an API to perform the most common solver operations
    - Can provide a single interface to many different solvers
    - Thyra: a collection of abstract interfaces to vectors, linear operators, linear solvers, etc.
    - TSFExtended: builds on top of TSFCore, providing implicit aggregation capabilities and overloaded operators
  - Petra provides common concrete implementations of basic classes used by most Trilinos packages
    - Supports the construction and use of vectors, sparse graphs, dense and sparse matrices
    - Provides serial and distributed memory (MPI) capabilities
    - Makes use of BLAS and LAPACK where possible
The **Trilinos** project

- **The Teuchos common tools package**
  - Templated access to BLAS and LAPACK
    - For data types single, double, complex single and complex double, the user is linked to standard BLAS and LAPACK functions
    - For other data types, provides generic loops sets for a limited set of key kernels
  - **Parameter lists**
    - A collection of key-value pairs that can be used to communicate with a package
    - Teuchos lists allow users to utilize the same parameter list constructs across multiple **Trilinos** packages
  - **Memory management tools** (allocation, deallocation etc.)
  - **Traits mechanisms**
    - Effective techniques for providing detailed information about supported generic data types
    - **ScalarTraits** define a variety of properties for supported scalar types (e.g. appropriate zero and one values for a given type)
    - **PacketTraits** is used to define the size of a packet type (allows generic use of data transfer algorithms e.g. via MPI)
  - Also: operation counts, exception handler and timers
The **Trilinos** project

- **Trilinos** package interoperability
  - Mechanism 1: package accepts user data as Epetra objects
    - All solver packages require some user data (vectors and matrices) or require the user to supply the action of an operator on a vector
    - Any package that accepts user data as Epetra objects becomes accessible to an application that has built its data using Epetra
  - Mechanism 2: package callable via TSF interfaces
    - TSF provides a set of abstract interfaces that can be used to interface to a variety of solver packages
    - TSF can accept pre-constructed solver objects (e.g. preconditioners, iterative solvers, etc.) by simple encapsulation or it can construct solver objects using one of a variety of factories
    - A package is callable via TSF if it implements one or more of the TSF abstract class interfaces
  - Mechanism 3: package can use Epetra internally
  - Mechanism 4: package accesses services via TSF interface
  - Mechanism 5: package builds under **Trilinos** configure scripts
    - **Trilinos** directory structure keeps each package completely self-contained
    - Each package is free to use its own configuration and build process
Working with Epetra vectors

- An Epetra vector may store either double precision values or integer values.
- An Epetra vector may be either serial or distributed (using a user-specified partitioning).
- The partitioning is based on:
  - a communicator,
  - a map (i.e., a partitioning of a list of global IDs).

The Epetra communicator objects

- *Epetra_Comp* virtual class with two basic concrete implementations
  - *Epetra_SerialComm* for serial executions
  - *Epetra_MpiComm* for MPI distributed memory executions

- Most *Epetra_Comp* methods are similar to MPI functions
  - *MyPID()*, *NumProc()*, *Barrier()*, *Broadcast()*, *SumAll()*, etc.
Working with Epetra vectors

Defining a map: the Epetra_Map class

- Distribution of a set of integer labels (or elements) across the processes
- Handles the definition of:
  - global number of elements in the set (called NumGlobalElements),
  - local number of elements (called NumMyElements),
  - global numbering of all local elements (an integer vector of size NumMyElements called MyGlobalElements).

- Offers a variety of methods for manipulating maps

Maps can be defined in 3 ways

- Specify the global number of elements and let Epetra define the distribution
  Epetra_Map Map(NumGlobalElements, 0, Comm);
- Give the local number of elements
  Epetra_Map Map(-1, NumMyElements, 0, Comm);
- Specify on each process both the local number of elements and the global indexing of each local element
  Epetra_Map Map(-1, NumMyElements, MyGlobalElements, Comm);

- For 1st and 2nd options, each process is assigned a contiguous set of elements
The Trilinos project

- Working with Epetra vectors
  - Creating and assembling a serial vector
    - Real-valued double precision dense vector
      ```
      Epetra_SerialDenseVector DoubleVector(Length);
      ```
    - Integer-valued dense vector
      ```
      Epetra_SerialIntVector IntVector(Length);
      ```
    - Other constructors are available
  - Creating and assembling a distributed vector
    - A distributed vector is created from a map
      ```
      Epetra_Vector x(Map);
      ⇒ Allocate space for the vector and set all elements to zero
      ```
    - Copy constructor: `Epetra_Vector y(x);`
    - A variety of sophisticated constructors are available
      ```
      Epetra_Vector x(Copy, Map, LocalValues);
      ⇒ Allocate space for the vector and copy the user-provided data
      ```
  - Setting the values of vector elements
    - Locally owned elements: `x[i] = 1.0*i;`
    - Epetra also provides functions to set elements in local or global index space
**The Trilinos project**

- Working with Epetra vectors
  - Inter-process communications: the Epetra_Import and Epetra_Export classes
    - Used to compute a communication plan that can be called repeatedly by computational classes
    - Both classes have one constructor that takes two maps as arguments:
      - 1st map specifies the global IDs of elements that are owned by the calling process,
      - 2nd map specifies the global IDs of elements that will be imported later.
    - Using an Epetra_Import object means that the calling process knows what it wants to receive
    - Using an Epetra_Export object means that the calling process knows what it wants to send
Working with Epetra vectors: a sample code
The Trilinos project

- Working with Epetra matrices
  - An Epetra matrix may be either serial or distributed
  - For distributed sparse matrices, the basic Epetra class is Epetra_RowMatrix (virtual class)
  - Derived classes:
    - Epetra_CrsMatrix for point matrices
    - Epetra_VbrMatrix for block matrices
    - Epetra_FECrsMatrix and Epetra_FEVbrMatrix for matrices arising from finite element discretizations
  - It is also possible to define serial dense matrices
    - Epetra_SerialDenseMatrix D(m, n);
    - Stored in a column-major order (as with Fortran)
    - This class is built on top of the BLAS library
    - Access elements using D(i,j) or D[j][i]
    - Solve a dense linear system with the class Epetra_SerialDenseSolver (built on top of BLAS and LAPACK)
Working with Epetra matrices

General procedure for constructing a (distributed) sparse matrix

1. Allocate an integer array Nnz whose length equals the number of local rows
2. Loop over the local rows and estimate the number of nonzero elements of that row
3. Create the sparse matrix using Nnz
4. Fill the sparse matrix

Sparse matrix arising from the finite-difference approximation of a 1D Laplace problem

- Set global dimension: `int NumGlobalElements = 5;`
- Create a map: `Epetra_Map Map(NumGlobalElements, 0, Comm);`
- Define local number of rows: `int NumMyElements = Map.NumMyElements();`
- Define global numbering for each local row
  ```c
  int *MyGlobalElements = Map.MyGlobalElements();
  ```
- Specify the number of nonzero elements per row: first solution
  ```c
  Epetra_CrsMatrix A(Copy, Map, 3);
  ```
  Here, Epetra considers the number 3 as a suggestion (can be higher in practice at the price of a possible performance decay)
Working with Epetra matrices

Sparse matrix arising from the finite-difference approximation of a 1D Laplace problem

Specify the number of nonzero elements per row: second solution

```c
int *NumNz = new int[NumMyElements];
for (int i=0; i<NumMyElements; ++i)
    if (MyGlobalElements[i] == 0 || MyGlobalElements[i] == NumGlobalElements-1)
        NumNz[i] = 2;
    else
        NumNz[i] = 3;
Epetra_CrsMatrix A(Copy, Map, NumNz);
```

Fill the matrix

For each local row, enter values and global column indices

Then, transform the matrix representation into one based on local indexes
The Trilinos project

Working with Epetra matrices: a sample code
The **Trilinos** project

- **Working with Epetra matrices**
  - The call to `FillComplete()` will reorganize the internally stored data so that each process knows the set of internal, border and external elements for performing a matrix-vector product
  - Also, the communication pattern is established
  - **Block matrices**
    - For PDE problems with more than one unknown per grid node
    - Block size of element $i$
      ```cpp
      Epetra_IntSerialDenseVector ElementSizeList(NumMyElements);
      for (int i=0; i<NumMyElements; ++i)
        ElementSizeList[i] = n_i;
      ```
    - Create a map and the associated block matrix
      ```cpp
      Epetra_BlockMap BlockMap(NumGlobalElements, NumMyElements,
                               MyGlobalElements,
                               ElementSizeList.Values(), 0, Comm);
      Epetra_VbrMatrix A(Copy, BlockMap, 2);
      ```

S. Lanteri (INRIA Sophia Antipolis)
The **Trilinos** project

- **Working with Epetra matrices**
  - The case of FE matrices: inserting non-local elements
  - Example: inserting diagonal entries of a matrix from process 0

```cpp
Epetra_FECrsMatrix A(Copy, Map, 1);
if (Comm.MyPID() == 0) {
    for (int i=0; i<NumGlobalElements; ++i) {
        int Indices[2];
        indices[0] = i; indices[1] = i;
        double value = 1.0*i;
        A.SumIntoGlobalValues(1, indices, &values);
    }
}
```

- Exchange data so that each matrix element not owned by process 0 is sent to the owner as specified by the map `Map`
  ```cpp
  A.GlobalAssemble();
  ```
The **Trilinos** project

- **Epetra linear problem**
  - A linear system $AX = B$ is defined by an `Epetra_LinearProblem` class
  - $X$ and $B$ are (multi-)vectors
  - Available methods
    - `void SetOperator(Epetra_RowMatrix *A)`
    - `void SetLHS(Epetra_MultiVector *X)`
    - `void SetRHS(Epetra_MultiVector *B)`
    - `Epetra_RowMatrix *GetMatrix() const`
    - `bool IsOperatorSymmetric() const`
    - etc.

- **Benchmarking the solution of a linear system: the Triutils package**
  - Predefined problem types (Laplace 1D/2D/3D, 2D flow model, etc.)
  - Include a command line parser
The **Trilinos** project

**Iterative solution of linear systems with AztecOO**

- A is a square, real coefficients, matrix
- Basic usage
  1. create a linear problem: `Epetra_LinearProblem Problem(&A, &x, &b);`
     (x and b are Epetra vectors; A is an Epetra matrix)
  2. create an AztecOO object: `AztecOO Solver(Problem);`
  3. specify how to solve the linear system: AztecOO options are set using two vectors

```c
int options[AZ_OPTIONS_SIZE];
double params[AZ_PARAMSS_SIZE];
AZ_defaults(options, params);
Solver.SetAllAztecOptions(options);
Solver.SetAllAztecParams(params);
```

4. solve the linear system: `Solver.Iterate(1000, 1.0e-09);`

- Matrix must be in local form (i.e. `A.FillComplete()` must have been invoked prior solving the linear system)
- Results: `Solver.NumIters()` and `Solver.TrueResidual()`
The Trilinos project

- Iterative solution of linear systems with AztecOO
  - Available solvers
    - CG: AZ_cg
    - CG + condition number estimation: AZ_cg_condnum
    - Restarted GMRES: AZ_gmres
    - Restarted GMRES + condition number estimation: AZ_gmres_condnum
    - CGS: AZ_cgs
    - Transpose-free QMR: AZ_tfqmr
    - BiCGStab: AZ_bicgstab
    - Serial sparse direct solver: AZ_lu
The **Trilinos** project

- Iterative solution of linear systems with AztecOO
  - Overlapping domain decomposition preconditioners
    - One level preconditioner: $P = \sum_{i=1}^{M} R_i^T B_i^{-1} R_i$
    - $B_i$ approximates $R_i A R_i^T$
      (e.g. when $B_i^{-1}$ is based on an incomplete factorization)
  - Basic usage
    ```
    Solver.SetAztecOption(AZ_precond, AZ_dom_decomp);
    Solver.SetAztecOption(AZ_subdomain_solve, AZ_ilu);
    Solver.SetAztecOption(AZ_graph_fill, 1);
    Solver.SetAztecOption(AZ_overlap, 1);
    ```

- Default overlap is 0 (block Jacobi)
- Other local solvers are available (including LU)
Parallel preconditioning with IFPACK

Provides a suite of object-oriented algebraic preconditioners

Overlapping (one-level) Schwarz methods: \( P = \sum_{i=1}^{M} P_i A_i^{-1} R_i \)

Local solves:
- one or more steps of a point/block method (Jacobi/Gauss-Seidel),
- ICC or ILU incomplete factorizations,
- CC or LU exact factorizations.

Supports an arbitrary level of overlap

Incomplete Cholesky factorizations

Compute the factorization of an Epetra_CrsMatrix

```c
Ifpack_CrsIct *ICT = NULL;
ICT = Ifpack_CrsIct(A, DropTol, LevelFill);
ICT->InitValues(A);
ICT->Factor();
```
The Trilinos project

- Incomplete factorizations with IFPACK
  - RILUK factorizations for non-symmetric matrices
  - Relaxed ILU: modified ILU factorization based on the strategy adopted for handling the dropped values
  - Factorization is split into two parts
    1. definition of a level filled graph
    2. computation of the factors
  - Well suited to the computation of several ILU factors based on the same sparsity pattern

\[
\text{Ifpack\_IlukGraph \ Graph()} = \text{Ifpack\_IlukGraph(A\_Graph, LevelFill, LevelOverlap)};
\]

- LevelOverlap is the required overlap among the subdomains
- A call to ConstructFilledGraph completes the process
- Create an Ifpack\_CrsRiluk object: \( \text{ILUK} = \text{Ifpack\_CrsRiluk(Graph)}; \) 
  \( \Rightarrow \) Defines the graph for the incomplete factorization
- Compute the actual values of the factors: \( \text{int initerr} = \text{ILUK} - > \text{InitValues}(A); \)
- Use the ILUK object with AztecOO: \( \text{Solver\_SetPrecOperator(ILUK)}; \)
Multilevel preconditioners with ML

- Application currently limited to certain elliptic PDEs
- Ongoing developments for incompressible Navier-Stokes equations and Maxwell’s equations
- Based on geometric and algebraic coarsening schemes
- Basic V-cycle scheme $\text{MGM}(x, b, K)$

\[
\begin{align*}
\text{IF } k > 0 & \\
& x = S_k^{(1)}(x, b) \\
& d = R_k^{k-1}(b - A_kx) \\
& v = 0 \\
& \text{MGM}(v, d, k - 1) \\
& x = x + P_k^{k-1}v \\
& x = S_k^{(2)}(x, b) \\
\text{ELSE} & \\
& x = A_k^{-1}b
\end{align*}
\]
The Trilinos project

- Multilevel preconditioners with ML
  - As with other Trilinos packages, ML can be compiled and run independently from Epetra
  - ML can also be used to define a preconditioner operator for an Epetra::LinearProblem object
  - This can be done in two ways:
    - by defining an ML::Epetra::MultiLevelOperator object,
    - by defining an ML::Epetra::MultiLevelPreconditioner object.
Multilevel preconditioners with ML

- The `ML_Epetra::MultiLevelOperator` object is derived from the `Epetra::Operator` class
  - Can be used to define geometric and algebraic multilevel preconditioners
  - Requires a deeper knowledge of the ML package (the user has to explicitly define all the components of the multilevel preconditioner)

- The `ML_Epetra::MultiLevelPreconditioner` object is derived from the `Epetra::RowMatrix` class
  - Limited to algebraic multilevel preconditioners
  - Automatically construct all the components of the multilevel preconditioner from a user given parameter list
The Trilinos project

- Multilevel preconditioners with ML
  - ML objects as AztecOO preconditioner
    - ML may be used as a black-box multilevel preconditioners
    - Multilevel hierarchy is defined using aggregation techniques
    - A ML preconditioner relies on a `ml_handle` structure to store internal data

```c
ML *ml_handle;
int N_levels = 10;
ML_Set_PrintLevel(3);
ML_Create(&ml_handle, N_levels);
```

- Construct an ML preconditioner for an Epetra matrix

```c
EpetraMatrix2MLMatrix(ml_handle, 0, &A);
ML_Aggregate *agg_object;
ML_Aggregate_Create(agg_object);
N_Levels = ML_Gen_MGHierarchy_UsingAggregation(ml_handle, 0,
                                              ML_INCREASING,
                                              agg_object);
```

- 0 is the index of the finest level
- `N_Levels` on output is the actual number of levels
The Trilinos project

- Multilevel preconditioners with ML
  - ML objects as AztecOO preconditioner
    - Define the smoother (e.g symmetric Gauss-Seidel)
      
      ```
      ML_Gen_Smoother_SymGaussSeidel(ml_handle, ML_ALL_LEVELS, ML_BOTH, 1, ML_DEFAULT);
      ```
  
  - Initialize the solver
    
    ```
    ML_Gen_Solver(ml_handle, ML_MGV, 0, N_Levels-1);
    ```
  
  - Create an Epetra Operator, set the preconditioner and iterate
    
    ```
    ML_Epetra::MultiLevelOperator MLop(ml_handle, com, map, map);
    Solver.SetPrecOperator(&Mlop);
    Solver.Iterate(Niters, 1.0e-12);
    ```
The Trilinos project

Related links

- Web page of Trilinos
  http://trilinos.sandia.gov/

- Epetra

- AztecOO
  http://trilinos.sandia.gov/packages/aztecoo/

- IFPACK
  http://trilinos.sandia.gov/packages/ifpack/
Some generalities

Overview of some freeware packages
- PETSc
- Trilinos
- Hypre

A concrete example
- The PORFLOW™ CFD software
- Improving PORFLOW™ performances
- Results
Generalities

- An object-oriented library for the solution of large sparse linear systems on parallel computers
- Designed by a team of the CASC (Center for Applied Scientific Computing) at the Lawrence Livermore National Laboratory
  - http://www.llnl.gov/CASC/linear_solvers/
- Latest release of the library: 2.0.0 (December 2006)
- Latest beta release: 2.2.0b (September 2007)
- The mathematical emphasis of Hypre is on modern powerful and scalable preconditioners:
  - parallel algebraic multigrid method (BoomerAMG, V.E. Henson and U.M. Yang, 2000),
  - sparse approximate inverse method with a priori sparsity patterns (ParaSails, E. Chow, 2000),
  - problem specific methods.
- Intended to be used by application developers as well as solver designers
Using **Hypre**

- **Specification step**
  1. Choose a *conceptual interface*
  2. Choose a solver/preconditioner
  3. Choose a matrix type that is compatible with:
     - the selected conceptual interface,
     - the selected solver/preconditioner.

- **Development step**
  1. Build auxiliary structures (e.g. grids, stencils, etc.)
  2. Build matrices and vectors through conceptual interface
  3. Build solver/preconditioner
  4. Solve the system
  5. Get desired information from the solver
Multiple interfaces are necessary to provide best solvers and data layouts

Key points:

- provide natural views of the linear system,
- ease some of the coding burden for users by eliminating the need to map to rows/columns,
- provide for more efficient (scalable) linear solvers,
- provide for more effective data storage schemes and more efficient computational kernels.

Two types of user interfaces for the definition of linear systems (matrices and vectors) and their solution:

- a classical linear-algebraic interface: the matrix is defined on a coefficient basis (the non-zero entries are passed through the interface for each row of the matrix);
- several physics-based interfaces: structured grids, semi-structured grids (block structured grids), finite element grids.
HYPRE conceptual interfaces

Linear System Interfaces

Linear Solvers

Data Layout
HYPRE conceptual interfaces

- The linear-algebraic interface: IJ
  - Matrices and the RHS are defined in terms of row and column indices
  - Data is expected in distributed form
    - A matrix is distributed by contiguous blocks of rows
  - More appropriate to linear system solver designers than to application developers

- The physics-based interfaces
  - Rationale: as problems grow in size and difficulty, it becomes increasingly necessary for solvers to have more information about the problem than what is encapsulated in the traditional matrix
  - Provide a mechanism by which information other than just the matrix can be passed into solvers
  - Linear systems are described in the language of the underlying problem domain
HYPRE conceptual interfaces

- The structured grid interface: **Struct**
  - Applications with logically rectangular grids
  - Appropriate for scalar applications on structured grids with a fixed stencil pattern
  - Grids are described via a global d-dimensional index space (singles in 1D, tuples in 2D and triples in 3D)
  - A **box** is a collection of cell-centered indices described by its lower and upper corners
The semi-structured grid interface: SStruct

- Applications with grids that are mostly (but not entirely) structured, e.g. block-structured, structured AMR, etc.
- Allows more general PDEs:
  - multiple variables (system of PDEs);
  - multiple variable types (cell centered, face centered, vertex centered, etc.).

- A SStruct grid is composed out of a number of structured grid parts
- The interface uses a graph to allow nearly arbitrary relationships between parts
- The graph is constructed from stencils plus some additional data-coupling information
HYPRE conceptual interfaces

The linear-algebraic interface: IJ

The IJ interface provides access to general sparse matrix solvers, but not specialized solvers.

Setting up the matrix (5 points stencil 2D Laplacian on a 10x10 grid)

HYPRE_IJMatrix A;
HYPRE_ParCSRMatrix parcsr_A;
int nrows = 3; int ncols[3] = {3, 4, 4}; rows[3] = {0, 1, 2};
int cols[11] = {0, 1, 10, 0, 1, 2, 11, 1, 2, 3, 12};
double values[11] = {4.0, -1.0, -1.0, -1.0, 4.0, -1.0, -1.0, -1.0, 4.0, -1.0, -1.0};
HYPRE_IJMatrixCreate(MPI_COMM_WORLD, ilower, iupper, jlower, jupper, &A);
HYPRE_IJMatrixSetObjectType(A, HYPRE_PARCSR);
HYPRE_IJMatrixInitialize(A);
/* Set matrix coefficients several rows at a time */
HYPRE_IJMatrixSetValues(A, nrows, ncols, rows, cols, values);
HYPRE_IJMatrixAssemble(A);
HYPRE_IJMatrixGetObject(A, (void **) &parcsr_A);
The linear-algebraic interface: \texttt{IJ}

- **Setting up the RHS vector**
  ```
  HYPRE_IJVector b;
  HYPRE_ParVector par\_b;
  int jlower, jupper, nvalues = 100;
  int indices[100] = \{0, 1, 2, ..., 97, 98, 99\};
  double values[100] = \{1.0, 1.0, ..., 1.0, 1.0\};
  HYPRE_IJVectorCreate(MPI\_COMM\_WORLD, jlower, jupper, &b);
  HYPRE_IJVectorSetObjectType(b, HYPRE\_PARCSR);
  HYPRE_IJVectorInitialize(b);
  /* Set several coefficients at a time */
  HYPRE_IJVectorSetValues(b, nvalues, indices, values);
  HYPRE_IJVectorAssemble(b);
  HYPRE_IJVectorGetObject(b, (void **) &par\_b);
  ```
## Hypre solvers

<table>
<thead>
<tr>
<th>Solvers</th>
<th>System interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Struct</td>
</tr>
<tr>
<td>Jacobi</td>
<td>X</td>
</tr>
<tr>
<td>SMG</td>
<td>X</td>
</tr>
<tr>
<td>PFMG</td>
<td>X</td>
</tr>
<tr>
<td>SysPFMG</td>
<td></td>
</tr>
<tr>
<td>Split</td>
<td></td>
</tr>
<tr>
<td>FAC</td>
<td></td>
</tr>
<tr>
<td>Maxwell</td>
<td></td>
</tr>
<tr>
<td>BoomerAMG</td>
<td>X</td>
</tr>
<tr>
<td>AMS</td>
<td>X</td>
</tr>
<tr>
<td>MLI</td>
<td>X</td>
</tr>
<tr>
<td>ParaSails</td>
<td>X</td>
</tr>
<tr>
<td>Euclid</td>
<td></td>
</tr>
<tr>
<td>PILUT</td>
<td></td>
</tr>
<tr>
<td>PCG</td>
<td>X</td>
</tr>
<tr>
<td>GMRES</td>
<td>X</td>
</tr>
<tr>
<td>BiCGStab</td>
<td>X</td>
</tr>
<tr>
<td>Hybrid</td>
<td>X</td>
</tr>
</tbody>
</table>
Setup and use of solvers is largely the same

- Create the solver
  
  \[
  \text{HYPRE\_SolverCreate}(\text{MPI\_COMM\_WORLD, } \&\text{solver});
  \]

- Set parameters
  
  \[
  \text{HYPRE\_SolverSetTol}(\text{solver, 1.0e-06});
  \]

- Prepare to solve the system
  
  \[
  \text{HYPRE\_SolverSetup}(\text{solver, A, b, x});
  \]

- Solve the system
  
  \[
  \text{HYPRE\_SolverSolve}(\text{solver, A, b, x});
  \]

- Get solution information out via conceptual interface
  
  \[
  \text{HYPRE\_StructVectorGetValues}(\text{struct\_x, index, values});
  \]

- Destroy solver
  
  \[
  \text{HYPRE\_SolverDestroy}(\text{solver});
  \]
SMG and PFMG are semicoarsening multigrid methods for structured grids

- Interfaces: Struct, SStruct
- SMG uses plane smoothing in 3D (each plane solve if effected by one 2D V-cycle)
- PFMG uses simple pointwise smoothing

SysPFMG is a semicoarsening multigrid method for systems of elliptic PDEs

- Interface: SStruct
- Nodal type relaxation

FAC is a fast adaptive composite grid solver

- Interface: SStruct
- For finite volume, cell-centered discretizations of smooth diffusion coefficient problems
Hybrid is a solver that allows for an adaptive switch between a diagonally scaled Krylov solver and a multigrid preconditioner

- Interface: Struct, SStruct, FEI and IJ
- Monitors how fast is the convergence of the diagonally scaled Krylov solver using an average convergence factor

Maxwell is a solver for edge finite element discretization of the curl-curl formulation of the Maxwell equations

- Interface: SStruct
- Operator dependent multiple coarsening algorithm for the Helmholtz decomposition of the error correction

AMS (Auxiliary space Maxwell Solver) is a parallel unstructured Maxwell solver for edge finite element discretizations

- Interface: SStruct, FEI and IJ
- Designed to be scalable on problems with variable coefficients
Hypre solvers

- MLI implements the class of algebraic multigrid algorithms based on the smoothed aggregation method
  - Interface: SStruct, FEI and IJ

- BoomerAMG is an AMG method for unstructured grids
  - Interfaces: SStruct, FEI and IJ
  - Uses simple pointwise smoothing

- ParaSails is a SPAI method
  - Interfaces: SStruct, FEI and IJ
  - Approximates the inverse of $A$ by a sparse matrix $M$ by minimizing the Frobenius norm $\|I - AM\|$
  - Uses graph theory to predict good sparsity patterns for $M$

- Euclid is a family of incomplete LU methods for sparse linear systems
  - Interfaces: SStruct, FEI and IJ
  - Obtains scalable parallelism via local and global reorderings
Hypre solvers: ILU factorization

- **Standard ILU(0)** (no fill-in) incomplete factorization
  - $A$ is a $n \times n$ matrix
  - $\text{NZ}(A)$ denotes the set of pairs $(i, j)$ such that $a_{i,j} \neq 0$
  - $a_{i,*}$ is the $i$th row of $A$
  - Find $U$ and $L$ such that $A = LU + E$ where $E$ is some error matrix
  - Purely *symbolic* factorization

```plaintext
1 FOR $i = 2, \cdots, n$ DO
  1 Define $u_{i,*} = a_{i,*}$
  2 FOR $k = 1, \cdots, i - 1$ and IF $(i, k) \in \text{NZ}(A)$ DO
    1 Compute the pivot $l_{i,k} = \frac{u_{i,k}}{u_{k,k}}$
    FOR $j = k + 1, \cdots, n$ and IF $(i, j) \in \text{NZ}(A)$ DO
      1 Compute $u_{i,j} = u_{i,j} - l_{i,k} u_{k,j}$
    3 ENDFOR
  3 ENDFOR
2 ENDFOR
```
Hypre solvers: ILU factorization

- Standard ILU(0) (no fill-in) incomplete factorization
  - Need for more accurate incomplete factorizations to improve convergence
  - ILU(p): level p fill-in $\Rightarrow$ drop fill-in elements whose levels are $> p$
  - For general matrices, the size of an element is not necessarily related to its level of fill-in
Generic \textbf{ILUT}(p, \tau) incomplete factorization

1. Set $r = 0$ ($r$ is a vector of size $n$)
2. FOR $i = 2, \cdots, n$ DO
   1. Sparse copy $r = a_{i,*}$
   2. FOR $k = 1, \cdots, i - 1$ and IF $r_k \neq 0$ DO
      Compute $r_k = \frac{r_k}{a_{k,k}}$
      Apply a dropping rule to $r_k$
      IF $r_k \neq 0$ THEN
         FOR $j = k + 1, \cdots, n$ DO
            $r_j = r_j - r_k u_{k,j}$
         ENDFOR
      ENDIF
   ENDFOR
3. ENDFOR
4. Apply a dropping rule to $r$
5. FOR $j = 1, \cdots, i - 1$ DO
   $l_{i,j} = r_j$
 ENDFOR
6. FOR $j = i, \cdots, n$ DO
   $u_{i,j} = r_j$
 ENDFOR
7. Set $r = 0$
3. ENDFOR
Generic \textbf{ILUT}(p, \tau) incomplete factorization

- 2.b.ii $\Rightarrow r_k$ is replaced by 0 is $r_k < \tau_i$ where $\tau_i = \tau \| a_{i,*} \|_2$
- 3. $\Rightarrow$ (1) apply the previous rule to each element of $r$ and (2) keep the $p$ largest elements in the $L$ and $U$ parts of the corresponding row (in addition to the diagonal element)
The first AMG (Algebraic MultiGrid) methods were initially developed in the 1980’s.

An AMG method is a hierarchical linear solver that works independently of any discretization grid (i.e. only the matrix operator $A$ is given).

The corresponding linear system may result from a problem defined on a structured grid or an unstructured grid as well.

In AMG methods, the coarse grid levels are subsets of the original set of unknowns:

- original grid: $\Omega = \{1, 2, \cdots, n\}$,
- a coarse grid is identified by a subset of indices of the unknown vector $u$ ($A^1 = A$ and $\Omega^1 = \Omega$).
Currently, there is a great interest in finding effective ways of applying AMG to extremely large problems involving millions of unknowns.

Much of the interest in AMG comes from the hope that the scalability of geometric multigrid methods can be obtained for large unstructured grid problems.

- AMG theory is well established for elliptic problems.
- Numerical efficiency for non-symmetric systems?
- Parallelization of AMG is not an easy task.
  - Parallel efficiency degrades when the number of coarse grid levels increases.
Hypre solvers: BoomerAMG

- Residual equation

\[ Ae = r \quad \text{with} \quad r = f - Au \]

- Basic process of any multigrid method (coarse grid correction scheme)
  - Damp the oscillatory components of the fine grid error \( e = u - u^{ex} \) through relaxation (i.e. smoothing),
  - Remove the smooth components of \( e \) by solving the residual equation on a coarser grid
  - Interpolate back the resulting error vector to the fine grid and use it to correct the fine grid approximation

- In AMG methods, the relaxation method is fixed and the main task is to determine a coarsening process that approximates well the error components the relaxation cannot reduce
The components of an AMG method

- A hierarchy of grid levels: $\Omega^1 \supset \Omega^2 \supset \cdots \supset \Omega^M$
- Grid operators: $A^1, A^2, \cdots, A^M$
- Intergrid transfer operators:
  - interpolation $I^k_{k+1}$ for $k = 1, 2, \cdots, M - 1$,
  - restriction $I^{k+1}_k$ for $k = 1, 2, \cdots, M - 1$.
- A relaxation scheme for each grid level
Multigrid V-cycle algorithm: $\text{MV}^k(u^k, f^k, \mu_1, \mu_2)$

\begin{align*}
\text{IF } k &= M \text{ THEN} \\
   u^M &= (A^M)^{-1} f^M \\
\text{ELSE} \\
   1. \text{ Relax } \mu_1 \text{ times on } A^k u^k = f^k \\
   2. \text{ Perform coarse grid correction:} \\
      &2.a \text{ Set } u^{k+1} = 0 \ , \ f^{k+1} = l_k^{k+1}(f^k - A^k u^k) \\
      &2.b \text{ Solve on level } k + 1 \text{ with } \text{MV}^{k+1}(u^{k+1}, f^{k+1}, \mu_1, \mu_2) \\
      &2.c \text{ Correct the solution } u^k \leftarrow u^k + l_{k+1}^k u^{k+1} \\
   3. \text{ Relax } \mu_2 \text{ times on } A^k u^k = f^k \\
\text{ENDIF}
\end{align*}
Hypre solvers: BoomerAMG

- Two basic principles
  
  **P1**: error components not efficiently reduced by relaxation must be well approximated by the range of interpolation
  
  **P2**: the coarse grid problem must provide a good approximation to fine grid error in the range of interpolation

- AMG satisfies **P1** by automatically selecting the coarse grid and defining interpolation, based solely on the system of algebraic equations

- **P2** is satisfied by defining restriction and the coarse grid operator through the Galerkin formulation:

  \[ I_{k+1}^k = (I_{k+1}^k)^T \quad \text{and} \quad A_{k+1}^k = I_{k+1}^k A_k I_{k+1}^k \]
The definition of some of the components of AMG is done within a setup phase:

- coarse grid levels,
- intergrid transfer operators.

1. Set $k = 1$
2. Partition $\Omega^k$ into disjoint sets $C^k$ and $F^k$
   - Set $\Omega^{k+1} = C^k$
   - Define interpolation $I_{k+1}^k$
3. Set $I_{k+1}^{k+1} = (I_{k+1}^k)^T$ and $A^{k+1} = I_{k+1}^{k+1} A^k I_{k+1}^k$
4. If $\Omega^{k+1}$ is small enough, set $M = k + 1$ and stop, otherwise set $k = k + 1$ and go to step 2
Step 2. is the core of the AMG setup phase.

The goal is to choose the set $C$ of coarse grid points and, for each fine grid point $i \in F \equiv \Omega - C$, a small set $C_i \subset C$ of interpolating points.

Definition of the interpolation operator:

\[
(I^k_{k+1} u^{k+1})_i = \begin{cases} 
    u_i^{k+1} & \text{if } \quad i \in C \\
    \sum_{j \in C_i} \omega_{ij} u_j^{k+1} & \text{if } \quad i \in F
\end{cases}
\]
HYPRE solvers: BoomerAMG

- The selection of coarse grid points consists in searching those unknowns $u_i$ which can be used to represent the values of nearby unknowns $u_j$.

- A point $i$ is said to depend on the point $j$ if the value of the unknown $u_j$ is important in determining the value of $u_i$ from the $i$th equation of the system $Au = f$.

- The set of dependences
  \[ S_i \equiv \{ j \neq i : -a_{ij} \geq \alpha \max_{k \neq i}(-a_{ik}) \} \]

- The set of influences
  \[ S_i^T \equiv \{ j : i \in S_j \} \]
A basic premise of AMG is that relaxation smooths the error in the direction of influence \( \Rightarrow \) select \( C_i = S_i \cap C \) as the set of interpolating points for \( i \).

In practice, the construction of \( C \) and \( F \) is subjected to two criteria:

\[ \text{C1} : \text{for each } i \in F, \text{ each } j \in S_i \text{ is either in } C \text{ or } S_j \cap C_i \neq 0 \]

\[ \text{C2} : C \text{ should be a maximal subset with the property that no point in } C \text{ depends on another point in } C \]

\( \text{C1} \) is used to insure that the value of \( u_j \) is represented in the interpolation formula for \( u_i \) if \( i \) strongly depends on \( j \).

\( \text{C2} \) is designed to strike a balance on the size of the coarse grid.
AMG employs a two-pass process for the construction of the $F$ and $C$ sets

**First pass**

1. Assign to each point $i$ the number $\lambda_i$ of other points strongly influenced by $i$
2. Select a point with maximal $\lambda$ as the first point in $C$
3. Mark the points that depend strongly on this $C$ point as $F$ points
4. For each new $F$ point $j$ in $S_i^T$, the $\lambda_k$ of points $k$ that are unassigned members of $S_j$ are incremented
5. Repeat 2. to 4. until all points are either $C$ or $F$ points

**In the second pass**, some $F$ points may be recolored as $C$ points in order to ensure that criterion $C1$ is satisfied
HYPRE solvers: BoomerAMG

- select C-pt with maximal measure
- select neighbors as F-pts
- update measures of F-pt neighbors

- select next C-pt with maximal measure
- select neighbors as F-pts
- update measures of F-pt neighbors
HYPRE solvers: BoomerAMG

- select next C-pt with maximal measure
- select neighbors as F-pts
- update measures of F-pt neighbors

- select next C-pt with maximal measure
- select neighbors as F-pts
- update measures of F-pt neighbors

- select next C-pt with maximal measure
- select neighbors as F-pts
- update measures of F-pt neighbors

- select next C-pt with maximal measure
- select neighbors as F-pts
- update measures of F-pt neighbors
HYPRE solvers: BoomerAMG

- select next C-pt with maximal measure
- select neighbors as F-pts
- update measures of F-pt neighbors

- select next C-pt with maximal measure
- select neighbors as F-pts
- update measures of F-pt neighbors
**Parallelization in the setup phase**

- Selection of the coarse gid points $\Omega^{k+1}$
- Construction of interpolation $l_{k+1}^k$ and restriction $l_{k}^{k+1}$
- Construction of the coarse grid operator: $A^{k+1} = l_{k}^{k+1} A^k l_{k+1}^k$

**Parallelization in the solve phase**

- Relaxation on $A^k u^k = f^k$
- Calculation of the residual: $r^k \leftarrow f^k - A^k u^k$
- Restriction of the residual: $f^{k+1} = l_{k}^{k+1} r^k$
- Interpolation and correction of the solution: $u^k \leftarrow u^k + l_{k+1}^k u^{k+1}$
Distributed memory programming model

- Most of the setup and solve steps can be easily parallelized
- Problem with the classical approach for the construction of the $C$ and $F$ sets (*inherently sequential process*)

**Solution**: associate to each point $i$ the measure $w(i) = |S_i^T| + \sigma(i)$ (i.e the number of points influenced by $i$ plus a random number in $(0, 1)$) and modify the first pass as:

1. Find a point $j$ with maximal $w(j)$ and select $j$ as a $C$ point
2. Designate neighbors of $j$ as $F$ points and update the measures of other neighbours using heuristics to insure grid quality
3. Repeat 1. and 2. until all points are either $C$ or $F$ points

⇒ Update of measures $w$ occurs after each $C$ point is selected
The second pass of the classical AMG setup phase is essentially used to ensure that criterion $C_1$ is satisfied

$\Rightarrow$ Can be eliminated through a simple modification of step 2.

Parallelization of the construction of the $C$ and $F$ points through a one-pass strategy

$\Rightarrow$ Begin by performing step 1. globally, selecting a set of $C$ points denoted by $D$ and then, perform step 2. locally with each processor working on some portion of the set $D$

By using different criteria for selecting the set $D$ and various heuristics for updating the neighbors in 2., a family of algorithms may be developed
Parallel selection of a coarse grid in the AMG setup phase

1. Input the $n \times n$ matrix $A^k$ (level $k$)
2. Initializations
   1. $F = \emptyset$, $C = \emptyset$
   2. $\forall i \in \{1, \cdots, n\}$: $w(i) \leftarrow$ initial value
3. LOOP until $|C| + |F| = n$
   1. Select an independent set of points $D$
   2. $\forall j \in D$:
      $C = C \cup j$
      $\forall k$ in set local to $j$, update $w(k)$
      IF $w(k) = 0$ THEN $F = F \cup k$
4. END LOOP

- In practice, a point $j$ is placed in $D$ if $w(j) > w(k)$, $\forall k \in S_j \cap S_j^T$.
  $\Rightarrow$ This step can be done entirely in parallel provided each processor has access to the $w$ values for points with influences that cross its processor boundary.
- By construction, $D$ is an independent set.
Hypre solvers: BoomerAMG

- Auxiliary influence matrix: $S_{ij} = \begin{cases} 1 & \text{if } j \in S_i \\ 0 & \text{otherwise} \end{cases}$

- Make use of the directed graph of $S$: an edge directed from vertex $i$ to vertex $j$ exists only if $S_{ij} \neq 0$

- The update of $w(k)$ relies on heuristics to insure the quality of the coarse grid while controlling its size
  
  **H1**: values at $C$ points are not interpolated; hence, neighbors that influence a $C$ point are less valuable as potential $C$ points themselves
  
  **H2**: if $k$ and $j$ both depend on $c$, a given $C$ point, and $j$ influences $k$, then $j$ is less valuable as a potential $C$ point since $k$ can be interpolated from $c$
Implementation of the **H1** and **H2** heuristics

**H1**:
\[
\forall c \in D \\
\forall j | S_{cj} \neq 0 \\
\quad - w(j) \leftarrow w(j) - 1 \\
\quad - S_{cj} \leftarrow 0
\]

(each \(j\) that influences \(c\))
(decrement the measure)
(remove edge \(cj\) from the graph)

**H2**:
\[
\forall c \in D \\
\forall j | S_{jc} \neq 0 \\
\quad - S_{jc} \leftarrow 0 \\
\quad - \forall k | S_{kj} \neq 0 \\
\quad \text{IF } S_{kc} \neq 0 \text{ THEN} \\
\quad \quad w(j) \leftarrow w(j) - 1 \\
\quad \quad S_{kj} \leftarrow 0 \\
\quad \text{ENDIF}
\]

(each \(j\) that depends on \(c\))
(remove edge \(jc\) from the graph)
(each \(k\) that depends on \(j\))
(if \(k\) depends on \(c\))
(decrement the measure)
(remove edge \(kj\) from the graph)
Hypre solvers: ParaSails

- Sparse approximate inverse (SPAI) are relatively recent parallel preconditioning techniques (M. Grote and H.D. Simon, 1993 - M.J. Grote and T. Huckle 1997)
- Key features of SPAI methods: robust, inherently parallel, no breakdown (A nonsingular), ordering independent, effective on non-symmetric and ill-conditioned problems
- ParaSails is a SPAI method based on a priori sparsity patterns (E. Chow, 2000)
Basic approach:

- solve the (left) preconditioned system : $MAx = Mb$,
- the goal is to construct the preconditioner $M \approx A^{-1}$,
- for SPD systems $A^{-1} \approx G^T G$ where $G$ is a sparse lower triangular matrix approximating the inverse of the lower triangular Cholesky factor $L$ of $A$,
- consider parallel methods that construct $M$ (general case) or $G$ (SPD case) by minimizing the Frobenius norm of the residual matrices defined by $(I - MA)$ or $(I - GL)$,
- in the non-factorized case, the objective function can be minimized in parallel because it can be decoupled as:

\[ \| I - MA \|_F^2 = \sum_{i=1}^{n} \| e_i^T - m_i^T A \|_2^2 \]

where $e_i^T$ and $m_i^T$ are the $i$th rows of $I$ and $M$. 

S. Lanteri (INRIA Sophia Antipolis)
Hypre solvers: ParaSails

- Minimize in parallel $\| e_i^T - m_i^T A \|_2$ for $i = 1, \cdots, n$

- In order to find an economical approximation of $A^{-1}$ each row of $M$ is constrained to be sparse:
  - specify the non-zero entries in $M$ a priori before the minimizations,
  - or during the minimizations using an adaptive strategy.

- A priori patterns:
  - banded patterns for banded matrices,
  - patterns of $A, A^2 \cdots$ (and variants) for more general matrices.

- In ParaSails, patterns of powers of sparsified matrices:
  - look for patterns of $\tilde{A}^l$ where $\tilde{A}$ is a sparse version of $A$ (numerical dropping of small entries),
  - $l - 1$ is the level of the pattern.

- Right preconditioning: minimize $\| I - AM \|_F^2$
HYPRE solvers: ParaSails

- Factorized case
  - The pattern of $G$ should be chosen such that the pattern of $G^T G$ is close in some sense to a good pattern for $A^{-1}$.
  - For SPD problems, minimizing $\| I - GL \|_F^2$ can be done without knowing $L$ by solving the normal equations:
    \[ \{GLL^T\}_{ij} = \{L^T\}_{ij} \text{ for } (i, j) \in S_L \]
    where $S_L$ is a lower triangular non-zero pattern for $G$
    \(\Downarrow\)
    \[ \{\tilde{G}A\}_{ij} = I_{ij} \text{ for } (i, j) \in S_L \text{ with } \tilde{G} = D^{-1}G \text{ and } D = \text{diag}(L) \]
    since $\{L^T\}_{ij}$ is a diagonal matrix for $(i, j) \in S_L$.
  - Each row of of $\tilde{G}$ can be computed independently by solving a small SPD linear system.
  - Form of the preconditioned matrix: $GAG^T = D\tilde{G}A\tilde{G}^TD$
  - $D$ is not known $\Rightarrow$ choose $D$ such that $\text{diag}(GAG^T)$ is all ones.
Filtering: drop non-zero entries of $M$ or $G$ that are small in magnitude to reduce the cost of storing and multiplying by the preconditioner

Algorithm 1: non-factorized sparse approximate inverse method

1. Threshold $A$ to produce $\tilde{A}$
2. Compute the pattern $\tilde{A}^l$ for $M$
3. Compute the non-zero entries in $M$ by minimizing $\|I - MA\|_F^2$
4. Filtering: drop small entries in $M$

Algorithm 2: factorized sparse approximate inverse method

1. Threshold $A$ to produce $\tilde{A}$
2. Compute the pattern $\tilde{A}^l$ and let the pattern of $G$ be the lower triangular part of the pattern of $\tilde{A}^l$
3. Compute the non-zero entries in $G$
4. Filtering: drop small entries in $G$ an rescale
1 Some generalities

2 Overview of some freeware packages
   - PETSc
   - Trilinos
   - Hypre

3 A concrete example
   - The PORFLOW™ CFD software
   - Improving PORFLOW™ performances
   - Results
Content

1. Some generalities

2. Overview of some freeware packages
   - PETSc
   - Trilinos
   - Hypre

3. A concrete example
   - The PORFLOW™ CFD software
   - Improving PORFLOW™ performances
   - Results
ACRi is a company specialized in CFD software and consulting

Localized in the US (Bel Air, CA and Cincinnati, OH), France (Sophia Antipolis) and India (Pradesh)

ACRi develops and commercializes a family of versatile CFD software for a wide range of applications

- ANSWER™: general purpose CFD simulator
- PORFLOW™: ground water, porous and fractured media simulator
- TIDAL™: oceanographic and surface water simulator
- RADM™: atmospheric pollution and transport simulator

PORFLOW™ is used by ANDRA for large-scale simulations of underground flows and contaminant transport

ANSWER™ and PORFLOW™ share the same linear solvers (NSPCG, University of Texas at Austin, 1998)
The PORFLOW™ CFD software

- PORFLOW™ can be used to solve various CFD problems:
  - transient or steady state fluid flow,
  - heat, salinity and mass transport,
  - in multi-phase, variably saturated, porous or fractured media with dynamic phase change.

- The porous/fractured media may be anisotropic and heterogeneous
- Arbitrary sources (injection or pumping wells) may be present
- Chemical reactions or radioactive decay may take place
- Accommodates alternate fluid and media property relations, and complex arbitrary boundary conditions
- The geometry may be 2D or 3D, cartesian or cylindrical and the mesh may be structured or unstructured
The PORFLOW™ CFD software

- **PORFLOW™ discretization method**
  - Nodal Point Integration method (Runchal, 1987, finite-volume type method)
  - Preserves the mass, material and thermal fluxes both at local and global scales
  - Appropriate integration (first and second order) profiles to insure stability and accuracy
  - Profiles correspond to central and upwind differencing
  - Profiles are combined in a hybrid scheme
  - The hybrid scheme automatically shift to one on the two profiles according to the value of the local Peclet number
  - Convective and diffusive terms are treated in a unified manner
The PORFLOW™ CFD software

- **PORFLOW™ time integration method**
  - \( \theta \)-scheme (explicit, semi-implicit or fully implicit)
  - In the implicit case, one algebraic equation for each dependent variable (i.e. scalar matrices)
  - Linear solvers:
    - ADI (both structured and unstructured),
    - SOR and SSOR,
    - preconditioned Krylov methods from the NSPCG package.

- ADI, SOR and SSOR are inlined in the PORFLOW™ code
- Access to NSPCG solvers and preconditioners relies on a simple interface module:
  - CG, CGNR, CGS, GMRES, etc.,
  - polynomial preconditioners,
  - incomplete factorization methods (ICC\((k)\) and ILU\((k)\)).
The COUPLEX activity

- Nuclear waste disposal simulations
  - Uses simplified albeit realistic test cases aimed at simulating the transport of radionuclides around a nuclear waste repository
  - COUPLEX1: 2D model
    - Is related to simulations based on a simplified 2D far-field model close to those used for safety assessments in nuclear waste management
    - It leads to a classical convection diffusion type problem, but with highly variable parameters in space, highly concentrated sources in space and time, very different time scales
  - COUPLEX2: 3D model
    - Is a simplification of a typical 3D near-field computation
    - Takes into account the glass dissolution of vitrified waste, and the congruent release of several radionuclides with their migration through the geological barrier
  - COUPLEX3: coupled model
    - Uses the results of the near-field computation (COUPLEX 2) to drive the behavior of the nuclide source term in the far-field computation (COUPLEX 1)
The COUPLEX1 (2D) test case

- Non-uniform structured quadrilateral grids

<table>
<thead>
<tr>
<th>Grid</th>
<th># nodes</th>
<th># elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1</td>
<td>187 × 111 = 20,757</td>
<td>20,165</td>
</tr>
<tr>
<td>G2</td>
<td>249 × 226 = 55,328</td>
<td>43,472</td>
</tr>
</tbody>
</table>

- Linear solvers
  - Calculation of the flow conditions: CG/ICC0 (with $\varepsilon_f = 10^{-6}$)
  - Transport of radioactive elements: ADI solver or preconditioned CGNR method (with $\varepsilon_t = 10^{-14}$)

- Simulation times (Intel Pentium P3/933 MHz)

<table>
<thead>
<tr>
<th>Test case</th>
<th>G1 (ADI)</th>
<th>G2 (ADI)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iodine 129</td>
<td>45 mn</td>
<td>330 mn</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solver</th>
<th>CGNR/NEU3</th>
<th>CGNR/ILU0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iodine 129, grid G2</td>
<td>620 mn</td>
<td>190 mn</td>
</tr>
</tbody>
</table>
The COUPLEX1 (2D) test case

- Effective number of CGNR/NEU3 and CGNR/ILU0 iterations

X-axis: time step
Y-axis: number of CGNR/NEU3 and CGNR/ILU0 iterations

Grid G1, $\| r \| = \| b - Ax \| < \varepsilon_t$
The COUPLEX1 (2D) test case

Non-uniform structured quadrilateral grid (grid G1)
The COUPLEX1 (2D) test case

Contour levels of the pressure (grid G1)
The COUPLEX1 (2D) test case

Contour levels of the concentration of Iodine 129 (time $10^6$ years, grid G1)
The COUPLEX2 (3D) test case

- Non-uniform structured hexahedral grid with $N_n = 134,961$ nodes and $N_e = 126,736$ elements

[Diagram showing the computational domain with boundary conditions and symmetry planes.

\[
\begin{align*}
  u(-x0) &= -u(x0) \\
  v(-x0) &= -v(x0) \\
  w(-x0) &= w(x0) \\
  H(-x0) &= H(x0)
\end{align*}
\]
The COUPLEX2 (3D) test case

- Calculation of the flow conditions: CG/ICC0 (with $\varepsilon_f = 10^{-14}$ and 300 outer iterations)
- Simulation time (Intel Pentium P3/700 MHz)

<table>
<thead>
<tr>
<th></th>
<th>CPU total</th>
<th>Aggreg. lin. solv.</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>182 mn</td>
<td>172 mn</td>
<td>95%</td>
<td></td>
</tr>
</tbody>
</table>
The COUPLEX2 (3D) test case

Non-uniform hexahedral mesh (plane $X = 4.5$)
The COUPLEX2 (3D) test case

Contour lines of the pressure (plane $X = 4.5$)
The COUPLEX2 (3D) test case

Non-uniform hexahedral mesh (plane $Z = 0.0$)
The COUPLEX2 (3D) test case

Contour lines of the pressure (plane Z = 0.0)
1. Some generalities

2. Overview of some freeware packages
   - PETSc
   - Trilinos
   - Hypre

3. A concrete example
   - The PORFLOW™ CFD software
   - Improving PORFLOW™ performances
   - Results
Improving PORFLOW™ performances

- Aggregate linear system solution times generally represent between 60% to 90% of total simulation times

- COUPLEX 1 timings
  - Iodine 129 test case
  - Intel Pentium P3/933 MHz
  - ADI solver for the transport phase

<table>
<thead>
<tr>
<th>Grid</th>
<th>CPU total</th>
<th>Aggreg. lin. solv.</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1</td>
<td>20 mn</td>
<td>15 mn</td>
<td>75%</td>
</tr>
<tr>
<td>G2</td>
<td>111 mn</td>
<td>100 mn</td>
<td>91%</td>
</tr>
</tbody>
</table>

- Grid G1
- CGNR/NEU3 and CGNR/ILU0 solvers for the transport phase

<table>
<thead>
<tr>
<th>Solver</th>
<th>CPU total</th>
<th>Aggreg. lin. solv.</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>CGNR/NEU3</td>
<td>620 mn</td>
<td>615 mn</td>
<td>99%</td>
</tr>
<tr>
<td>CGNR/ILU0</td>
<td>190 mn</td>
<td>185 mn</td>
<td>97%</td>
</tr>
</tbody>
</table>
Goal: reduce linear system solution times
- Improved solvers and preconditioners
- Parallel computing

Candidate methods in the **sequential** case
- Incomplete factorization methods (dual threshold ILU)
- Algebraic multigrid methods

Candidate methods in the **parallel** case
- Parallel algebraic multigrid methods
- Sparse Approximate inverse methods
- Domain decomposition methods:
  - additive Schwarz type methods,
  - Schur complement type methods.

Selection criteria
- General purpose methods
  - Applicable to SPD as well as to general systems
  - Not restricted to a particular matrix structure
- Numerical efficiency and parallel efficiency
Goal: reduce linear system solution times

Adopted strategy

- Select methods from domain public linear algebra packages
- Evaluate the selected methods using representative linear systems that have been extracted from PORFLOW™ simulations

Linear algebra packages considered in this study:

- SparseKit (U. of Minnesota),
- PsparsLib (U. of Minnesota),
- Hypre (Lawrence Livermore National Laboratory).

Question: does the flexibility of (parallel) object oriented libraries also come with improved overall efficiency as compared to classical procedural (sequential) libraries?
The **SparseKit** package

- A basic toolkit for sparse matrix computations
  - Designed by Y. Saad (University of Minnesota, Department of Computer Science and Engineering) and collaborators
  - Latest version of the library: 2.0 (June 1994)
- Standard ILU(0) (no fill-in) incomplete factorization
  - $A$ is a $n \times n$ matrix
  - $NZ(A)$ denotes the set of pairs $(i, j)$ such that $a_{i,j} \neq 0$
  - $a_{i,*}$ is the $i$th row of $A$
  - Find $U$ and $L$ such that $A = LU + E$ where $E$ is some error matrix
  - Purely *symbolic* factorization

1. **FOR** $i = 2, \cdots, n$ **DO**
   1. Define $u_{i,*} = a_{i,*}$
   2. **FOR** $k = 1, \cdots, i - 1$ and **IF** $(i, k) \in NZ(A)$ **DO**
      1. Compute the pivot $l_{i,k} = \frac{u_{i,k}}{u_{k,k}}$
      2. **FOR** $j = k + 1, \cdots, n$ and **IF** $(i, j) \in NZ(A)$ **DO**
          - Compute $u_{i,j} = u_{i,j} - l_{i,k}u_{k,j}$
  3. **ENDFOR**
  2. **ENDFOR**
The **SparseKit** package

- Standard ILU(0) (no fill-in) incomplete factorization
  - Need for more accurate incomplete factorizations to improve convergence
  - ILU$(p)$ : level $p$ fill-in $\Rightarrow$ drop fill-in elements whose levels are $> p$
  - For general matrices, the size of an element is not necessarily related to its level of fill-in
The SparseKit package

- Generic ILUT\((p, \tau)\) incomplete factorization

1. Set \(r = 0\) (\(r\) is a vector of size \(n\))
2. FOR \(i = 2, \cdots, n\) DO
   1. Sparse copy \(r = a_{i,*}\)
   2. FOR \(k = 1, \cdots, i - 1\) and IF \(r_k \neq 0\) DO
      1. Compute \(r_k = \frac{r_k}{a_{k,k}}\)
      2. Apply a dropping rule to \(r_k\)
      3. IF \(r_k \neq 0\) THEN
         - FOR \(j = k + 1, \cdots, n\) DO \(r_j = r_j - r_k u_{k,j}\) ENDFOR
   3. ENDIF
   3. ENDFOR
3. Apply a dropping rule to \(r\)
4. FOR \(j = 1, \cdots, i - 1\) DO \(l_{i,j} = r_j\) ENDFOR
5. FOR \(j = i, \cdots, n\) DO \(u_{i,j} = r_j\) ENDFOR
6. Set \(r = 0\)
7. ENDFOR
The **SparseKit** package

- **Generic ILUT\((p, \tau)\)** incomplete factorization
  - Dropping rule on \(r_k \Rightarrow r_k\) is replaced by 0 if \(r_k < \tau_i\) where \(\tau_i = \tau \| a_i,\star \|_2\)
  - Dropping rule on \(r\) \(\Rightarrow\)
    - apply the previous rule to each element of \(r\)
    - keep the \(p\) largest elements in the \(L\) and \(U\) parts of the corresponding row (in addition to the diagonal element)
The PsparsLib package

- A portable library of parallel sparse iterative solvers
  - Designed by Y. Saad (University of Minnesota, Department of Computer Science and Engineering) and collaborators
  - Latest version of the library : 3.0 (November 1999)
  - A set of tools for solving large sparse linear systems on distributed-memory parallel computers:
    - accelerators (CG, BiCG, BiCGSTAB, GMRES, FGMRES),
    - preconditioning routines,
    - preprocessing tools and message-passing tools.

- Preconditioning techniques based on domain decomposition principles (overlapping and non-overlapping versions):
  - block Jacobi (additive Schwarz),
  - multicolor block SOR (multicolor multiplicative Schwarz),
  - Schur complement techniques.

- Flexibility is maximized by using a reverse communication mechanism:
  - the functional modules are independent from date structures and specifics of message-passing,
  - matrix-by-vector products and preconditioning operations: a functional routine returns to the calling program to let it perform the desired operation.
The \textbf{PsparsLib} package

- Solution of $Ax = b$ where $A$ is a large sparse matrix
- Assume the linear system results from a (finite difference, finite volume or finite element) discretization of a PDE
- Distributed sparse linear systems
  - Partitioning of the matrix $A$ and the vectors $x$ and $b$
    - by partitioning the underlying discretization grid (if available) and assigning the resulting submeshes to the processing nodes,
    - by assigning equation-unknown pairs to the processing nodes.
- In both cases, at the end of the distribution step, a processing node is responsible for a subset of the equations of the linear system
- The partitioning can be obtained from appropriate graph partitioning algorithms
- Goal: develop parallel solvers and parallel preconditioners for the solution of the global linear system using appropriate distributed data structures for the involved matrix and vectors
A local view of a distributed sparse matrix from the physical domain viewpoint of a sparse linear system as adopted in PsparsLib.
Distributed sparse linear system: the physical domain viewpoint

- A node corresponds to an equation-unknown pair
- For each subdomain $\Omega_i$, three types of unknown:
  - **internal unknowns** are associated to purely internal nodes of $\Omega_i$,
  - **local interface unknowns** are associated to local interface nodes of $\Omega_i$,
  - **external unknowns** are associated to external interface nodes of $\Omega_i$.

- Internal unknowns are only coupled (through $Ax = b$) to other internal unknowns of $\Omega_i$,
- Local interface unknowns are coupled to internal unknowns of $\Omega_i$ and also to external unknowns
- External unknowns are under the responsibility of subdomains $\Omega_j$ that are neighbors of $\Omega_i$ (i.e. external interface unknowns of $\Omega_i$ are local interface unknowns of $\Omega_j$)
The PsparsLib package

Local representation of a linear system

- Rows of matrix $A$ which are assigned to the processing node responsible for $\Omega_i$ are separated into two parts:
  - a **local matrix** $A_i$ which acts on internal unknowns (purely internal unknowns and local interface unknowns) of $\Omega_i$,
  - an **interface matrix** $X_i$ which acts on external interface unknowns of $\Omega_i$.

- External interface unknowns have to be received from the processing nodes responsible for the neighboring $\Omega_j$ prior to the computation of the local part of the matrix-vector product $(Ax)_i$.
Remark: the subset of equations associated to $\Omega_i$ do not necessarily correspond to contiguous equations of the global linear system.

A local vector of unknowns $x_i$ is separated in two parts:

- the subvector $u_i$ of purely internal unknowns,
- the subvector $y_i$ of local interface unknowns.

$$x_i = \begin{pmatrix} u_i \\ y_i \end{pmatrix}, \quad b_i = \begin{pmatrix} f_i \\ g_i \end{pmatrix}$$

Corresponding block partitioning of the matrix $A$

$$A_i = \begin{pmatrix} B_i & E_i \\ F_i & C_i \end{pmatrix}$$
The **PsparsLib** package

- **Local representation** of a linear system

\[
\begin{pmatrix}
B_i & E_i \\
F_i & C_i
\end{pmatrix}
\begin{pmatrix}
u_i \\
y_i
\end{pmatrix}
+ 
\begin{pmatrix}
0 \\
\sum_{j \in N_i} E_{ij}y_j
\end{pmatrix}
= 
\begin{pmatrix}
f_i \\
g_i
\end{pmatrix}
\]

- \(E_{ij}y_j\) is the contribution of \(\Omega_j\) to the equations associated to the local interface nodes of \(\Omega_i\)

- \(N_i\) is the set of neighboring subdomains of \(\Omega_i\)

\[
\sum_{j \in N_i} E_{ij}y_j \equiv X_iy_{i,\text{ext}}
\]
The PsparsLib package

- **Additive Schwarz preconditioner**
  - The simplest domain decomposition preconditioner
  - Algebraic domain decomposition methods are iterative methods based on a partitioning of the system $Ax = b$
  - The partitioning can be geometric or purely algebraic
  - Preconditioners based on domain decomposition principles are essentially block preconditioners (block = subdomain)
  - The basic iteration can be cast as a succession of resolutions of local residual equations $r_i = b_i - (Ax)_i$

- **Basic steps**
  1. Communication of the unknowns associated to external interface nodes $y_{i,\text{ext}}$
  2. FOR each subdomain $\Omega_i$ DO IN PARALLEL
     1. Compute the local residual $r_i = (b - Ax)_i = b_i - A_i x_i - X_i y_{i,\text{ext}}$
     2. Solve $A_i \delta_i = r_i$
     3. Update the local solution $x_i = x_i + \delta_i$
  3. END DO
The PsparsLib package

- Additive Schwarz preconditioner
  - Local solves: ILUT($p, \tau$) preconditioned GMRES method or one step of a classical ILU($p$) incomplete factorization method
  - Accelerator (global solver): FGMRES
The **PsparsLib** package

- **Schur complement type techniques**
  - The main iteration acts on the unknowns associated to nodes located on the interfaces between neighboring subdomains.
  - Implicit use of unknowns associated to purely internal nodes as intermediate variables.
  - Classical versions use a non-overlapping partition.
  - **PsparsLib** adopts a general strategy for deriving Schur complement techniques from an arbitrary global fixed point iteration (eventually based on an overlapping partition).
    - Eliminate purely internal unknowns from the local linear system
      \[ u_i = B_i^{-1}(f_i - E_i y_i) \]
    - Substitute in the second part of local equations
      \[ S_i y_i + \sum_{j \in N_i} E_{ij} y_j = g_i - F_i B_i^{-1} f_i \]
The PsparsLib package

- Schur complement type techniques
  - Local Schur complement operator
    \[ S_i = C_i - F_i B_i^{-1} E_i \]
  - Assembling of local systems yield the formulation of an interface system
    \[ Sy = g' \]
  - Vector of interface unknowns \( y = (y_1, y_2, \cdots, y_{s-1}, y_s)^T \)
  - \( s \) is the total number of subdomains
  - The diagonal blocks of \( S \) are given by the dense matrices \( S_i \)
  - The extradiagonal blocks \( E_{ij} \) are sparse matrices

- With a consistent choice of the initial guess, a block Jacobi iteration on the interface system \(\Leftrightarrow\) a block Jacobi iteration on the system
  - Corresponding iteration for the vector of interface unknowns \( y_i \)
    \[ y_i^{(k+1)} = S_i^{-1} \left[ g_i - F_i B_i^{-1} f_i - \sum_{j \in N_i} E_{ij} y_j^{(k)} \right] \]
The **PsparsLib** package

\[
x_{i}^{(k+1)} = x_{i}^{(k)} + A_{i}^{-1}r_{i}^{(k)}
\]

\[
= x_{i}^{(k)} + A_{i}^{-1}\begin{pmatrix} b_{i} - A_{i}x_{i}^{(k)} - \left( \begin{array}{c} 0 \\ \sum_{j \in N_{i}} E_{ij}y_{j}^{(k)} \end{array} \right) \end{pmatrix}
\]

\[
= A_{i}^{-1}\begin{pmatrix} f_{i} \\ g_{i} - \sum_{j \in N_{i}} E_{ij}y_{j}^{(k)} \end{pmatrix}
\]

\[
= \begin{pmatrix} * & * \\ -S_{i}^{-1}F_{i}B_{i}^{-1} & S_{i}^{-1} \end{pmatrix}\begin{pmatrix} f_{i} \\ g_{i} - \sum_{j \in N_{i}} E_{ij}y_{j}^{(k)} \end{pmatrix}
\]

\[
x_{i}^{(k)} = \begin{pmatrix} u_{i}^{(k)} \\ y_{i}^{(k)} \end{pmatrix}, \quad b_{i} = \begin{pmatrix} f_{i} \\ g_{i} \end{pmatrix}
\]
The PsparsLib package

- Schur complement type techniques
  - ⋅ is used for terms that are not relevant for the demonstration
  - Global viewpoint
    - Primary iteration: $x^{(k+1)} = Mx^{(k)} + c$
    - Associated iteration on interface unknowns: $y^{(k+1)} = Gy^{(k)} + h$
  - Remarks:
    - the matrix $M$ is easily deduced from the matrix $A$ of the global linear system,
    - the matrix $G$ is not explicitly known,
    - it is easy to compute the matrix-vector product $Gv$ for a given vector $v$,
    - the iteration on $y$ can also be interpreted as a block Jacobi method for the system $(\text{Id} - G)y = h$,
    - the solution of this system can be accelerated using GMRES.
    - the matrix-vector product $w = (\text{Id} - G)y$ can be obtained from the original iteration on $x$. 

S. Lanteri (INRIA Sophia Antipolis)
The **PsparsLib** package

- **Schur complement type techniques**
  - **Matrix-vector product** \( w = (\text{Id} - G)y \)
    - Communication of the unknowns associated to external interface nodes \( y_{i,\text{ext}} \)
    - Perform one iteration of the primary iteration using the vector \( (0, y)^T \) as an initial condition:
      \[
      \begin{pmatrix}
      u' \\
      y'
      \end{pmatrix} = M \begin{pmatrix} 0 \\ y \end{pmatrix} + c
      \]
    - Define \( w := y' \)
    - Compute \( W' := y - w + h \)

  \( \Rightarrow \) Easy construction of a Schur complement type method from a primary iteration (block Jacobi or block Gauss-Seidel method)
Content

1. Some generalities

2. Overview of some freeware packages
   - PETSc
   - Trilinos
   - Hypre

3. A concrete example
   - The PORFLOW™ CFD software
   - Improving PORFLOW™ performances
   - Results
Results

- The COUPLEX1 (2D) test case
  - Non-uniform structured quadrilateral grids
  - Matrices for grid G1
    - G1-h:
      - flow calculation,
      - pentadiagonal matrix,
      - $n_z = 101,507$.
    - G1-t:
      - transport calculation, - pentadiagonal matrix, - $n_z = 122,990$.
  - Matrices for grid G2
    - G2-h:
      - flow calculation,
      - pentadiagonal matrix,
      - $n_z = 277,534$.
    - G2-t:
      - transport calculation, - pentadiagonal matrix, - $n_z = 333,968$. 
Results

- 3D transport calculation
- Non-structured hexahedral grid with 247,936 elements
  - U250–0: the initial hydraulic conditions are characterized by a zero Darcy velocity in all the computational domain
    - \( n_z = 1,724,640 \) (SPD matrix)
  - U250–V: the initial hydraulic conditions are characterized by a non-zero Darcy velocity but still constant in all the computational domain
    - \( n_z = 1,719,212 \)

- Reference solvers (from NSPCG) and solution times (Intel Pentium P3/933 MHz)

<table>
<thead>
<tr>
<th>Case</th>
<th>Solver/Preconditioner</th>
<th>( N_{\text{iter}} )</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>U250–0</td>
<td>CG/ICC0</td>
<td>65</td>
<td>47 sec</td>
</tr>
<tr>
<td>U250–V</td>
<td>GMRES(5)/NEU3</td>
<td>164</td>
<td>222 sec</td>
</tr>
</tbody>
</table>

S. Lanteri (INRIA Sophia Antipolis)
Results

- 3D transport calculation
- Non-structured hexahedral grid with 247,936 elements
The COUPLEX1 (2D) test case: flow calculations (G1-h and G2-h)

- HP i-cluster of Intel Pentium P3/733 MHz, Ethernet 100 Mbit/s
- Reference times: NSPCG, CG/ICC0: 9 sec (G1-h) and 39 sec (G2-h)
- AMG characteristics:
  - strength threshold: $\alpha = 0.25$,
  - hybrid Falgout-CLJP coarsening,
  - hybrid Gauss-Seidel/Jacobi smoother (V(2,exact,2)-cycle).

<table>
<thead>
<tr>
<th>Package</th>
<th>Grid</th>
<th>Method</th>
<th>$N_{\text{lev}}$</th>
<th>$N_p$</th>
<th>$N_{\text{iter}}$</th>
<th>Elapse/CPU total</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>HYPRE</strong></td>
<td>G1-h</td>
<td>CG/AMG</td>
<td>10</td>
<td>1</td>
<td>12</td>
<td>2.0 sec</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>10</td>
<td>2</td>
<td>13</td>
<td>1.6 sec / 1.5 sec</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>10</td>
<td>4</td>
<td>12</td>
<td>1.5 sec / 1.3 sec</td>
</tr>
<tr>
<td>-</td>
<td>G2-h</td>
<td>AMG</td>
<td>15</td>
<td>1</td>
<td>26</td>
<td>9.0 sec</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>15</td>
<td>2</td>
<td>27</td>
<td>5.9 sec / 5.7 sec</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>15</td>
<td>4</td>
<td>28</td>
<td>5.3 sec / 4.2 sec</td>
</tr>
<tr>
<td>-</td>
<td>G2-h</td>
<td>CG/AMG</td>
<td>15</td>
<td>1</td>
<td>15</td>
<td>7.0 sec</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>15</td>
<td>2</td>
<td>15</td>
<td>4.6 sec / 4.4 sec</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>15</td>
<td>4</td>
<td>17</td>
<td>4.5 sec / 3.3 sec</td>
</tr>
</tbody>
</table>
The COUPLEX1 (2D) test case: flow calculations (G1-h and G2-h)

X-axis : time step  -  Y-axis : number of AMG or CG/AMG iterations
Results

- The COUPLEX1 (2D) test case: transport calculation for grid G2-t
  - Cluster of Intel Pentium P3/933 MHz, Ethernet 100 Mbit/s
  - Reference time: **NSPCG, GMRES(10)/ILU0**: 84 sec (226 iterations)
  - **PsparsLib**: FGMRES(10) with overlapping Schwarz (right) preconditioning
    - Local solves: ILUT ($l_{fil} = 15$ and $\varepsilon_{tol} = 10^{-4}$)

<table>
<thead>
<tr>
<th>Package</th>
<th>$N_{procs}$</th>
<th>$N_{iter}$</th>
<th>CPU total</th>
<th>CPU prec. (min/max)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PsparsLib</td>
<td>2</td>
<td>109</td>
<td>9.5 sec</td>
<td>3.5 sec/4.0 sec</td>
</tr>
<tr>
<td>-</td>
<td>4</td>
<td>127</td>
<td>5.5 sec</td>
<td>2.0 sec/3.0 sec</td>
</tr>
<tr>
<td>-</td>
<td>8</td>
<td>142</td>
<td>3.5 sec</td>
<td>1.0 sec/2.0 sec</td>
</tr>
</tbody>
</table>

S. Lanteri (INRIA Sophia Antipolis)
Results

- The COUPLEX1 (2D) test case: transport calculation for grid G2-t

![Graph showing time step vs. number of FGMRES iterations for different numbers of processes.]

X-axis: time step - Y-axis: number of FGMRES iterations
Results

- **U250-0 test case (transport by diffusion only)**
  - HP i-cluster of Intel Pentium P3/733 MHz, Ethernet 100 Mbit/s
  - Reference times: NSPCG, CG/ICC0 : 56 sec
  - **HYPRE**: CG (left) preconditioned by AMG
  - AMG characteristics:
    - strength threshold : $\alpha = 0.85$,
    - hybrid Falgout-CLJP coarsening,
    - hybrid Gauss-Seidel/Jacobi smoother (V(2,2,2)-cycle).

<table>
<thead>
<tr>
<th>Package</th>
<th>$N_{procs}$</th>
<th>$N_{iter}$</th>
<th>Elapse/CPU total</th>
<th>Elapse/CPU set-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>HYPRE</td>
<td>1</td>
<td>12</td>
<td>35 sec</td>
<td>19 sec</td>
</tr>
<tr>
<td>-</td>
<td>4</td>
<td>12</td>
<td>22 sec/21 sec</td>
<td>19 sec/16 sec</td>
</tr>
<tr>
<td>-</td>
<td>8</td>
<td>10</td>
<td>14 sec/11 sec</td>
<td>32 sec/15 sec</td>
</tr>
</tbody>
</table>

S. Lanteri (INRIA Sophia Antipolis)
Results

- **U250–0 test case (transport by diffusion only)**

X-axis: time step - Y-axis: number of CG/AMG iterations
Results

U250–0 test case (transport by diffusion only)

- HP i-cluster of Intel Pentium P3/733 MHz, Ethernet 100 Mbit/s
- Reference times: NSPCG, CG/ICC0: 56 sec
- Hypre: CG (left) preconditioned by SPAI
- SPAI characteristics:
  - factorized approximate inverse (A is SPD),
  - 2 levels and $\varepsilon_{thrsh} = 10^{-2}$.

<table>
<thead>
<tr>
<th>Package</th>
<th>$N_{procs}$</th>
<th>$N_{iter}$</th>
<th>Elapse/CPU total</th>
<th>Elapse/CPU set-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hypre</td>
<td>1</td>
<td>68</td>
<td>48 sec</td>
<td>18 sec</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>68</td>
<td>30 sec/ 29 sec</td>
<td>18 sec/17 sec</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>68</td>
<td>17 sec/ 16 sec</td>
<td>15 sec/14 sec</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>68</td>
<td>11 sec/ 9 sec</td>
<td>15 sec/14 sec</td>
</tr>
</tbody>
</table>
**Results**

- **U250-V test case (transport by convection-diffusion only)**
  - Intel Pentium P3/933 MHz
  - **SparseKit**: GMRES/ILUT($p, \tau$)

<table>
<thead>
<tr>
<th>Package</th>
<th>Solver/Preconditioner</th>
<th>$N_{iter}$</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>NSPCG</td>
<td>GMRES(5)/NEU3</td>
<td>164</td>
<td>222 sec</td>
</tr>
<tr>
<td>SparseKit</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>s1</td>
<td>58</td>
<td>61 sec</td>
</tr>
<tr>
<td></td>
<td>s2</td>
<td>32</td>
<td>53 sec</td>
</tr>
<tr>
<td></td>
<td>s3</td>
<td>19</td>
<td>43 sec</td>
</tr>
</tbody>
</table>

S. Lanteri (INRIA Sophia Antipolis)  
Linear algebra libraries for HPSC
• U250-V test case (transport by convection-diffusion only)

X-axis: time step - Y-axis: number of GMRES/ILUT iterations
Results

- U250-V test case (transport by convection-diffusion only)
  - Cluster of Intel Pentium P3/933 MHz, Ethernet 100 Mbit/s
  - Reference time: NSPCG, GMRES(5)/NEU3: 222 sec
  - PsparsLib: FGMRES(10) with overlapping Schwarz (right) preconditioning
    - Local solves: 5 iterations of GMRES(10)/ILUT ($l_{fil} = 15$ and $\varepsilon_{tol} = 10^{-4}$)

<table>
<thead>
<tr>
<th>Package</th>
<th>$N_{procs}$</th>
<th>$N_{iter}$</th>
<th>CPU total</th>
<th>CPU preconditioning (min/max)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PsparsLib</td>
<td>4</td>
<td>47</td>
<td>31 sec</td>
<td>19 sec/25 sec</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>48</td>
<td>18 sec</td>
<td>11 sec/14 sec</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>60</td>
<td>11 sec</td>
<td>5 sec/ 9 sec</td>
</tr>
</tbody>
</table>
Results

- U250-V test case (transport by convection-diffusion only)

X-axis : time step - Y-axis : number of FGMRES iterations

S. Lanteri (INRIA Sophia Antipolis)
Results

- **U250-V test case** (transport by convection-diffusion only)

  - Cluster of Intel Pentium P3/933 MHz, Ethernet 100 Mbit/s
  - Reference time: \textit{NSPCG, GMRES(5)/NEU3} : 222 sec
  - \texttt{PsparsLib}: FGMRES(10) with overlapping Schur complement (right) preconditioning

    - Local solves: 5 iterations of GMRES(10)/ILUT ($l_{fil} = 15$ and $\varepsilon_{tol} = 10^{-4}$)

<table>
<thead>
<tr>
<th>Package</th>
<th>$N_{\text{procs}}$</th>
<th>$N_{\text{iter}}$</th>
<th>CPU total</th>
<th>CPU preconditioning (min/max)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{PsparsLib}</td>
<td>2</td>
<td>17</td>
<td>17 sec</td>
<td>13 sec/13 sec</td>
</tr>
<tr>
<td>-</td>
<td>4</td>
<td>18</td>
<td>15 sec</td>
<td>12 sec/13 sec</td>
</tr>
<tr>
<td>-</td>
<td>8</td>
<td>19</td>
<td>14 sec</td>
<td>12 sec/13 sec</td>
</tr>
</tbody>
</table>
Results

- **U250-V test case (transport by convection-diffusion only)**

![Graph showing time step vs. number of FGMRES iterations for different PSPARSLIB processes](image)

- **X-axis:** time step
- **Y-axis:** number of FGMRES iterations
Results

- The COUPLEX1 (2D) test case (Iodine 129)
  - Intel Pentium P3/700 MHz
  - NSPCG solvers
    - Calculation of the flow conditions: CG/ICC0 (with $\varepsilon_f = 10^{-6}$)
    - Transport of radioactive elements: ADI (with $\varepsilon_t = 10^{-14}$)
    - Reference simulation times: 45 mn (grid G1) and 330 mn (grid G2)
  - Hypre solvers
    - Calculation of the flow conditions: CG/AMG
    - Transport of radioactive elements: GMRES(15)/ILUT(25,10$^{-3}$)

<table>
<thead>
<tr>
<th>Grid</th>
<th>CPU total</th>
<th>Aggreg. lin. solv.</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1</td>
<td>14 mn</td>
<td>9 mn</td>
<td>65%</td>
</tr>
<tr>
<td>G2</td>
<td>56 mn</td>
<td>38 mn</td>
<td>68%</td>
</tr>
</tbody>
</table>
Results

- The COUPLEX1 (2D) test case (Iodine 129)
  - Influence of the linear threshold $\varepsilon_t$ on the accuracy of the computed solution for the transport phase
  - Net flux disparity coefficient
    \[ Q_{\text{dis}} = Q_{\text{in}} - Q_{\text{out}} - (Q_n - Q_o) - Q_{\text{dec}} \]
    - $Q_o$: amount of component present originally
    - $Q_n$: amount of component present now
    - $Q_{\text{in}}$: cumulative total inflow
    - $Q_{\text{out}}$: cumulative total outflow
    - $Q_{\text{dec}}$: cumulative decay

<table>
<thead>
<tr>
<th>$\varepsilon_t$</th>
<th>CPU total</th>
<th>Aggreg. lin. solv.</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^{-14}$</td>
<td>56 mn</td>
<td>38 mn</td>
<td>68%</td>
</tr>
<tr>
<td>$10^{-15}$</td>
<td>60 mn</td>
<td>42 mn</td>
<td>70%</td>
</tr>
<tr>
<td>$10^{-16}$</td>
<td>65 mn</td>
<td>47 mn</td>
<td>73%</td>
</tr>
</tbody>
</table>

GMRES(15)/ILUT(25, $10^{-3}$), grid G2
The COUPLEX1 (2D) test case (Iodine 129)

- Influence of the linear threshold $\varepsilon_t$ on the accuracy of the computed solution for the transport phase
The COUPLEXD2 (3D) test case: calculation of the flow conditions

- Intel Pentium P3/700 MHz
- NSPCG solver
  - CG/ICC0 (with $\varepsilon_f = 10^{-14}$): 182 mn
- HYPRE solver, CG/AMG (with $\varepsilon_f = 10^{-14}$): 104 mn
Linear algebra frewares: the key points

- Offer access to modern solvers/preconditioners
  (in some cases, resulting from very recent research projects)
  - Sparse direct solvers
  - Algebraic multigrid methods
  - Sparse approximate inverse methods
  - Algebraic domain decomposition methods

- Are based on modern software development concepts
  - Object oriented programming
  - Interoperability mechanisms

- Ease the development of complex, large-scale, scientific computing applications
  - Access to algebraic as well as grid-oriented interfaces
  - Parallel computing transparently