Partial Differential Equations in HPC, II

M. D. Jones, Ph.D.

Center for Computational Research
University at Buffalo
State University of New York

High Performance Computing I, 2013

What We Have Done

- Direct Solvers
- Tridiagonal Systems
- Multigrid
- Initial-value Problems, linear and nonlinear
- Iterative Methods

What We Have Yet to Do

- Preconditioners
- Conjugate Gradient
- Adaptive Meshes
- Available Libraries

Jacobi Revisited

Recall Jacobi iterations in matrix form,

$$\mathbf{A}\mathbf{x} = \mathbf{b}$$

Now if we write the left-hand matrix **A** as a sum of its diagonal and non-diagonal parts,

$$\mathbf{A} = \mathbf{L} + \mathbf{D} + \mathbf{U}$$

and in this context we have,

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

So in the Jacobi scheme, the "iteration matrix" is given by:

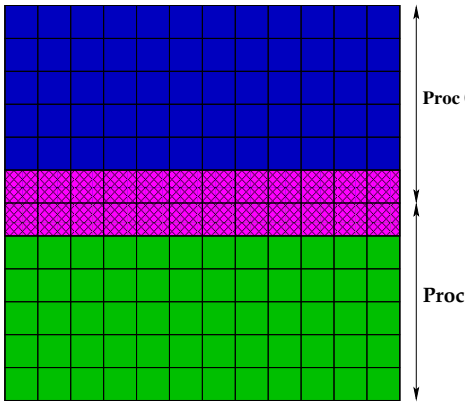
$$\mathbf{D}^{-1}(\mathbf{L}+\mathbf{U})$$

Written this way, it is not surprising to learn that the eigenvalues of this iteration matrix reflect suppression of the residual errors (and these factors must be < 1 to converge at all).

The **spectral radius** is just the modulus of the largest such factor, and is closely related the rate of convergence. For Jacobi, the number of iterations required is proportional to square of the number of grid points (very slow!).

Parallel Jacobi

We talked before about using data parallel methods for parallel Jacobi, but what about distributed parallel? It turns out that typically it is well worth doing a bit of additional computation to minimize communication. The “ghost” cell concept, is illustrated in the following figure:



where the green and blue regions represent decomposition on two processors, and the magenta the shared boundary upon which updates must be exchanged.

Parallel Jacobi (cont'd)

- Each processor can update N^2/P points (2D), note that each processor only has to store the region that it is responsible for after domain decomposition (decreases memory requirements)
- Amount of data communicated, N/P, relatively small for $N >> P$

Gauss-Seidel, Revisited

In matrix form, we can write Gauss-Seidel in a form similar to Jacobi,

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

It can be shown that G-S converges in about half the number of iterations as Jacobi (still too slow!). In parallel (e.g. red-black scheme) often twice as many steps, so just as bad as Jacobi. Use of updated information in G-S can be carried a step further still,

$$\mathbf{x}^{k+1} = \mathbf{x}^{k} + \omega \times \mathbf{correct}^{k}$$

and we can "overcorrect" in a further attempt to accelerate the convergence of the relaxation method.

Successive Over-Relaxation (SOR)

SOR is a variant on Gauss-Seidel, builds in a "mixing" of the old and updated solution:

$$(\omega \mathbf{L} + \mathbf{D})\mathbf{x}^{k+1} = -\left[\omega \mathbf{U} + (1-\omega)\mathbf{D}\right]\mathbf{x}^{k} + \omega \mathbf{b},$$

or expressed in elements using forward substitution:

$$x_i^{k+1} = (1-\omega)x_i^k + \frac{\omega}{a_{ii}}\left(b_i - \sum_{j>i} a_{ij}x_j^k - \sum_{j<i} a_{ij}x_j^{k+1}\right),$$

for symmetric, positive definite matrices, it can be shown that $0 < \omega < 2$ leads to convergence (but it does not guarantee convergence rate).

Successive Over-Relaxation (SOR), Revisited

- Successive Over-Relaxation (SOR)
- Parallelize like Jacobi (still sparse matrix-vector multiply)
- Standard until 1970s
- $0 < \omega < 2$ convergent
- $0 < \omega < 1$ under-relaxation
- $1 < \omega < 2$ generally faster than G-S
- Richardson extrapolation (generalized for any iterative method): for $\mathbf{x}^{k+1} = F(\mathbf{x}^k)$,

$$\mathbf{x}^{k+1} = \omega \mathbf{x}^k + (1 - \omega) F(\mathbf{x}^k).$$

SOR (cont'd)

- Can show best convergence for:

$$\omega = \frac{2}{1 + \sqrt{(1 - \rho_J^2)}}$$

  where ρ_J is the spectral radius of the Jacobi update
- Quite sensitive to ω, c.f. *Chebyshev acceleration*, in which ω is adjusted at half-cycles, itself iteratively improved (c.f. *Numerical Recipes*)
- Number of steps to converge $\mathcal{O}(N)$, much better! (Jacobi was $\mathcal{O}(N^2)$)

Convergence Rate Simplified

A detailed analysis of convergence properties is beyond our scope. For simple cases like our $M \times M$ grid in a 2D Laplace boundary value problem with Dirichlet boundary conditions[1], the iteration matrix for Jacobi's method has a spectral radius, ρ_J, estimated by:

$$\rho_J \simeq 1 - \frac{\pi^2}{2M^2},$$

and the number of iterations, R_J, to reduce the error by 10^{-x} is

$$R_J \simeq \frac{x \ln 10}{(-\ln \rho_J)} \simeq \frac{xM^2}{2},$$

the corresponding "optimal" SOR would have

$$\omega \simeq \frac{2}{1 + \pi/M},$$

$$\rho_{SOR} \simeq 1 - \frac{2\pi}{M},$$

$$R_{SOR} \simeq \frac{xM}{3}.$$

[1] again, *c.f. Numerical Recipes* and references therein

Conjugate Gradient

Conjugate Gradient (CG), one of the most popular sparse/implicit methods:

Quote

"How could fifteen lines of pseudocode take fifty pages to explain?"

- Initial guess, $\mathbf{x}^{(0)}$
- $\mathbf{r}^{(0)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)}$
- Repeat until ρ reaches small tolerance:
  - $\rho = \mathbf{r}^T\mathbf{r}$ (ddot)
  - if first iteration, $\mathbf{p} = \mathbf{r}$, else $\beta = \rho/\rho_{old}$ and $\mathbf{p} = \mathbf{r} + \beta\mathbf{p}$ (daxpy)
  - $\mathbf{q} = \mathbf{A}\mathbf{p}$ (matrix-vector)
  - $\alpha = \rho/\mathbf{p}^T\mathbf{q}$ (ddot)
  - $\mathbf{x} = \mathbf{x} + \alpha\mathbf{p}$ (daxpy)
  - $\mathbf{r} = \mathbf{r} - \alpha\mathbf{q}$ (daxpy)
  - $\rho_{old} = \rho$

Algorithm is pretty simple, underlying methodology requires some background ...

Steepest Descent

CG is based on the idea of steepest descent (SD), and the quadratic form

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x} - \mathbf{b}^{\mathrm{T}}\mathbf{x} + c,$$

for which $f(\mathbf{x})$ is minimized by the solution to $\mathbf{A}\mathbf{x} = \mathbf{b}$ for a matrix $\mathbf{A}$ which is symmetric and positive definite.

$$\mathbf{f}' = \begin{bmatrix} \partial f/\partial x_1 \\ \partial f/\partial x_2 \\ \vdots \\ \partial f/\partial x_N \end{bmatrix} = \mathbf{A}\mathbf{x} - \mathbf{b},$$

Steepest Descent (cont'd)

SD, then, as the name suggests takes a starting point $\mathbf{x}^{(0)}$, then takes successive steps in the direction in which f decreases the fastest, i.e.

$$-\mathbf{f}'(\mathbf{x}^{(i)}) = \mathbf{r}^{(i)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(i)},$$

and one chooses the step size, α, such that $\mathbf{f}$ is minimized, i.e. $\partial \mathbf{f}(\mathbf{x}^{i+1})/\partial\alpha = 0$, or

$$\begin{aligned}
-\mathbf{f}'(\mathbf{x}^{(i+1)})^{\mathrm{T}}\mathbf{r}^{(i)} &= 0, \\
(\mathbf{b} - \mathbf{A}\mathbf{x}^{(i+1)})^{\mathrm{T}}\mathbf{r}^{(i)} &= 0, \\
(\mathbf{b} - \mathbf{A}(\mathbf{x}^{(i)} + \alpha\mathbf{r}^{(i)})^{\mathrm{T}}\mathbf{r}^{(i)} &= 0, \\
(\mathbf{b} - \mathbf{A}\mathbf{x}^{(i)})^{\mathrm{T}}\mathbf{r}^{(i)} - \alpha(\mathbf{A}\mathbf{r}^{(i)})^{\mathrm{T}}\mathbf{r}^{(i)} &= 0, \\
\alpha &= \frac{(\mathbf{r}^{(i)})^{\mathrm{T}}\mathbf{r}^{(i)}}{(\mathbf{r}^{(i)})^{\mathrm{T}}\mathbf{A}\mathbf{r}^{(i)}}.
\end{aligned}$$

Steepest Descent Summary

Summary of SD:

- Choose starting point, $\mathbf{x}^{(0)}$,
- Run until convergence:

$$\mathbf{r}^{(i)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(i)},$$

$$\alpha^{(i)} = \frac{(\mathbf{r}^{(i)})^{\mathrm{T}}\mathbf{r}^{(i)}}{(\mathbf{r}^{(i)})^{\mathrm{T}}\mathbf{A}\mathbf{r}^{(i)}},$$

$$\mathbf{x}^{(i+1)} = \mathbf{x}^{(i)} + \alpha^{(i)}\mathbf{r}^{(i)},$$

It can be shown that steepest descent converges like $(\kappa - 1)/(\kappa + 1)$, where κ is the spectral condition number (ratio of largest to smallest eigenvalue of $\mathbf{A}$). This rate is very slow for poorly conditioned $\mathbf{A}$ or close to the solution.

SD Example: Rosenbrock Function

Consider a simple example in 2D, Rosenbrock function ($f(x, y) = (1 - x)^2 + 100(y - x^2)^2$, also applied in N dimensions) with SD:

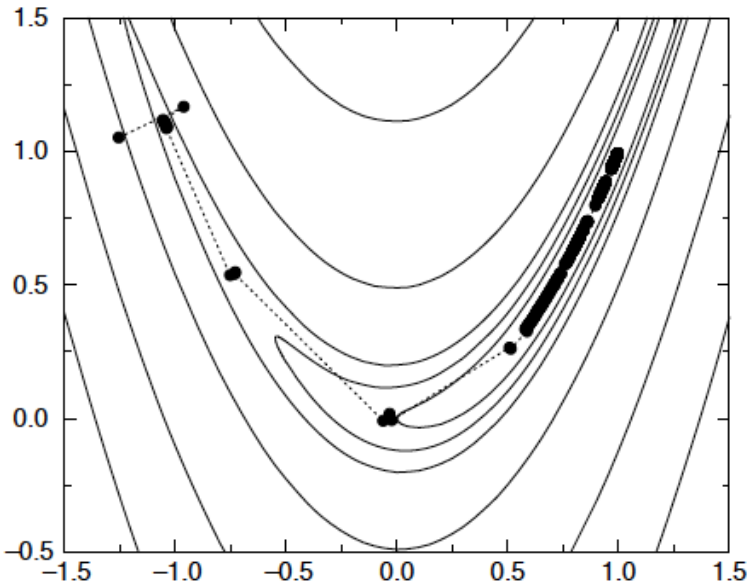


Figure 9: Steepest Descent Minimization Path for the Two-Dimensional Rosenbrock Function.

Stopped after 1200 iterations, achieved only 10^{-2} in gradient norm (from the Computational Science Education Project, Mathematical Optimization, p. 21).

Enter Conjugate Gradient

Conjugate gradient for $\mathbf{Ax} = \mathbf{b}$ builds into SD the additional idea of taking your steps orthogonal to the previous step. Some nomenclature:

- Recall that any two nonzero vectors, **u,v** are conjugate with respect to the matrix **A** if $\langle \mathbf{u}, \mathbf{v} \rangle_{\mathbf{A}} = \mathbf{u}^{\mathrm{T}} \mathbf{A} \mathbf{v} = 0$.
- Operator matrix must be symmetric ($\mathbf{A} = \mathbf{A}^T$) ...
- ...and positive definite:
  - Eigenvalues positive
  - $\mathbf{x}^T \mathbf{A} \mathbf{x} > 0$, for all nonzero vectors, **x**
  - Cholesky factorization, $\mathbf{A} = \mathbf{L}\mathbf{L}^T$, exists
- Basic idea is to minimize $\mathbf{Ax} - \mathbf{b} = \mathbf{0}$ using conjugate directions computed from **A**.

Conjugate Gradient (cont'd)

- CG maintains three vectors:
  - search direction, **p** (a.k.a the conjugate gradient)
  - approximate solution, **x**, improved iteratively
  - residual, **r**, defined by $\mathbf{r} = \mathbf{Ax} - \mathbf{b}$
- Each iteration costs one sparse matrix-vector multiply, plus 2 inner (dot) products and three `daxpy` (scale vector plus vector) operations

Conjugate Gradient (cont'd)

- Conjugate directions, $\mathbf{p}$, form a basis with respect to $\mathbf{A}$,

$$\mathbf{x} = \sum_{i=1}^{n} \alpha_i \mathbf{p}_i,$$

- How to find α_i?

$$\mathbf{b} = \mathbf{A}\mathbf{x} = \sum_{i=1}^{n} \alpha_i \mathbf{A}\mathbf{p}_i,$$

- Multiply by $\mathbf{p}_j^{\mathrm{T}}$:

$$\mathbf{p}_j^{\mathrm{T}}\mathbf{b} = \sum_{i=1}^{n} \alpha_i \mathbf{p}_j^{\mathrm{T}}\mathbf{A}\mathbf{p}_i = \alpha_j \mathbf{p}_j^{\mathrm{T}}\mathbf{A}\mathbf{p}_j,$$

- Which gives us α_j in terms of the conjugate directions:

$$\alpha_j = \frac{\mathbf{p}_j^{\mathrm{T}}\mathbf{b}}{\mathbf{p}_j^{\mathrm{T}}\mathbf{A}\mathbf{p}_j}$$

Conjugate Gradient (cont'd)

Basic CG algorithm in pseudo-code:

- Initial guess, $\mathbf{x}^{(0)}$
- $\mathbf{r}^{(0)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)}$
- Repeat until ρ reaches small tolerance:
  - $\rho = \mathbf{r}^T\mathbf{r}$ (ddot)
  - if first iteration, $\mathbf{p} = \mathbf{r}$, else $\beta = \rho/\rho_{old}$ and $\mathbf{p} = \mathbf{r} + \beta\mathbf{p}$ (daxpy)
  - $\mathbf{q} = \mathbf{A}\mathbf{p}$ (matrix-vector)
  - $\alpha = \rho/\mathbf{p}^T\mathbf{q}$ (ddot)
  - $\mathbf{x} = \mathbf{x} + \alpha\mathbf{p}$ (daxpy)
  - $\mathbf{r} = \mathbf{r} - \alpha\mathbf{q}$ (daxpy)
  - $\rho_{old} = \rho$

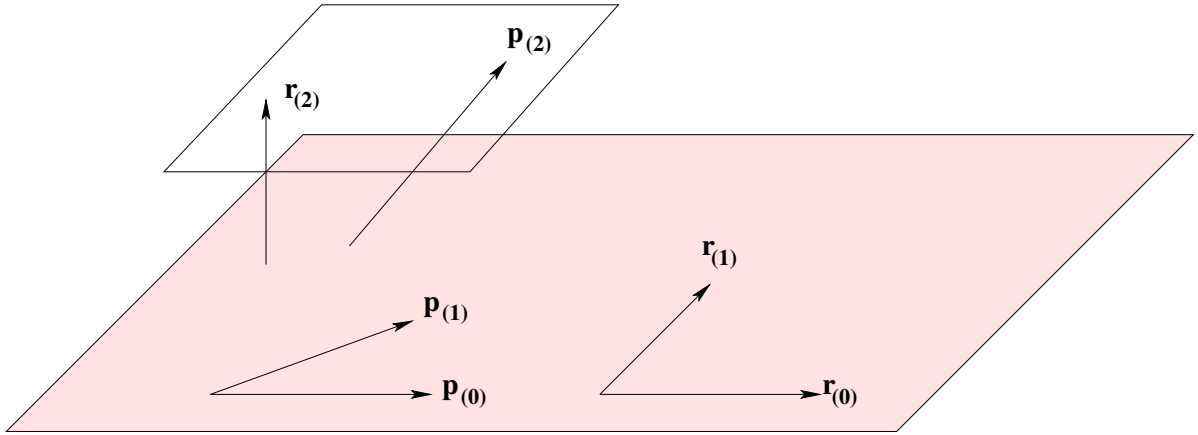
CG (cont'd)

Quote

"How could fifteen lines of pseudocode take fifty pages to explain?"

- J. Shewchuk "An Introduction to the Conjugate Gradient Method Without the Agonizing Pain," (1994)
  http://www.cs.cmu.edu/~jrs/jrspapers.html
- CG converges in $\mathcal{O}(N)$ steps, like SOR
- Each residual vector is orthogonal to previous choices (unless it is zero, in which case we are done)
- CG is an example of a *Krylov Subspace* method, namely the subspace spanned by the residual vectors is obtained from repeatedly applying the matrix **A**
- Major achievement is that space/time complexity effectively reduced from $\mathcal{O}(N^2)$ to $\mathcal{O}(m)$, where m is the number of nonzero elements of **A**

CG Illustrated



- Each new residual $\mathbf{r}_i$ is orthogonal to all previous residuals,
- Each new direction is **A**-orthogonal to previous residuals and search direction
- Endpoints of $\mathbf{r}_2$ and $\mathbf{p}_2$ lie on plane parallel to (shaded) subspace spanned by $\mathbf{r}_0$ and $\mathbf{r}_1$
- Convergence is even quicker for better conditioned **A**, for which we can use preconditioners ...

CG Example: Rosenbrock Function

Consider again the simple example in 2D, Rosenbrock function ($f(x, y) = (1 - x)^2 + 100(y - x^2)^2$, also applied in N dimensions) with CG this time:

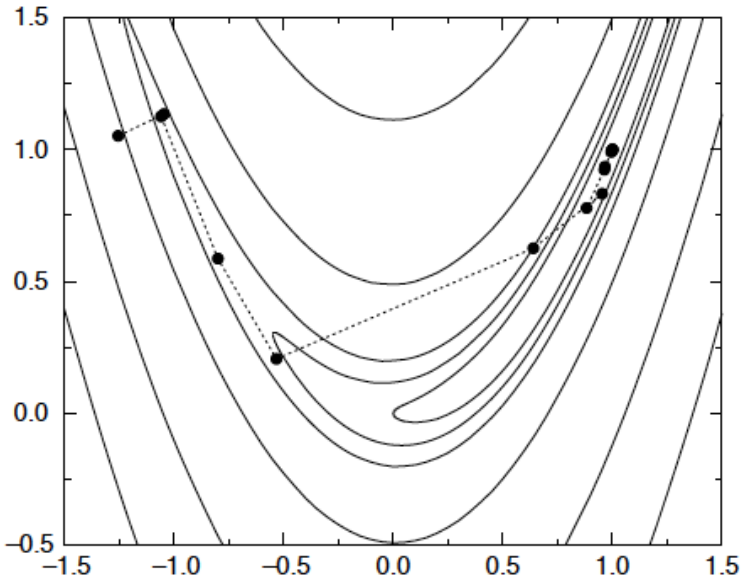


Figure 10: Conjugate Gradient Minimization Path for the Two-Dimensional Rosenbrock Function.

Now the gradient norm is 10^{-9} after only a dozen or so iterations. (from the Computational Science Education Project, Mathematical Optimization, p. 21)

Preconditioners

For iterative methods the convergence rate depends strongly on the spectral properties of the left-hand side (operator) matrix. If we transform the matrix:

$$\mathbf{M}^{-1}\mathbf{A}\mathbf{x} = \mathbf{M}^{-1}\mathbf{b},$$

(e.g. by choosing $\mathbf{M} \sim \mathbf{A}$) we may end up with a much faster rate of convergence. These matrices are called **preconditioners** and are much applied in CG and related techniques.

CG With Preconditioning

CG algorithm in pseudo-code, with preconditioning:

- Initial guess, $\mathbf{x}^{(0)}$
- $\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}$
- Repeat until ρ reaches small tolerance:
  - $\rho = \mathbf{r}^T\mathbf{r}$ (ddot)
  - if first iteration, $\mathbf{p} = \mathbf{M}^{-1}\mathbf{r}$, else $\beta = \rho/\rho_{old}$ and $\mathbf{p} = \mathbf{M}^{-1}\mathbf{r} + \beta\mathbf{p}$ (daxpy)
  - $\mathbf{q} = \mathbf{A}\mathbf{p}$ (matrix-vector)
  - $\alpha = \rho/\mathbf{p}^T\mathbf{q}$ (ddot)
  - $\mathbf{x} = \mathbf{x} + \alpha\mathbf{p}$ (daxpy)
  - $\mathbf{r} = \mathbf{r} - \alpha\mathbf{q}$ (daxpy)
  - $\rho_{old} = \rho$

Note that a common choice of preconditioner is just $\mathbf{M}$=`diag`($\mathbf{A}$).

Available Software for Solving PDEs

Not ready to reinvent the wheel? There are a number of available software libraries designed for various general classes of problems.

By no means are we going to try for an exhaustive survey - our focus will be on the largest, best supported (in terms of continued development and funding), and freely available packages.

Netlib Listing

A good place to start is the following list compiled by Jack Dongarra:

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

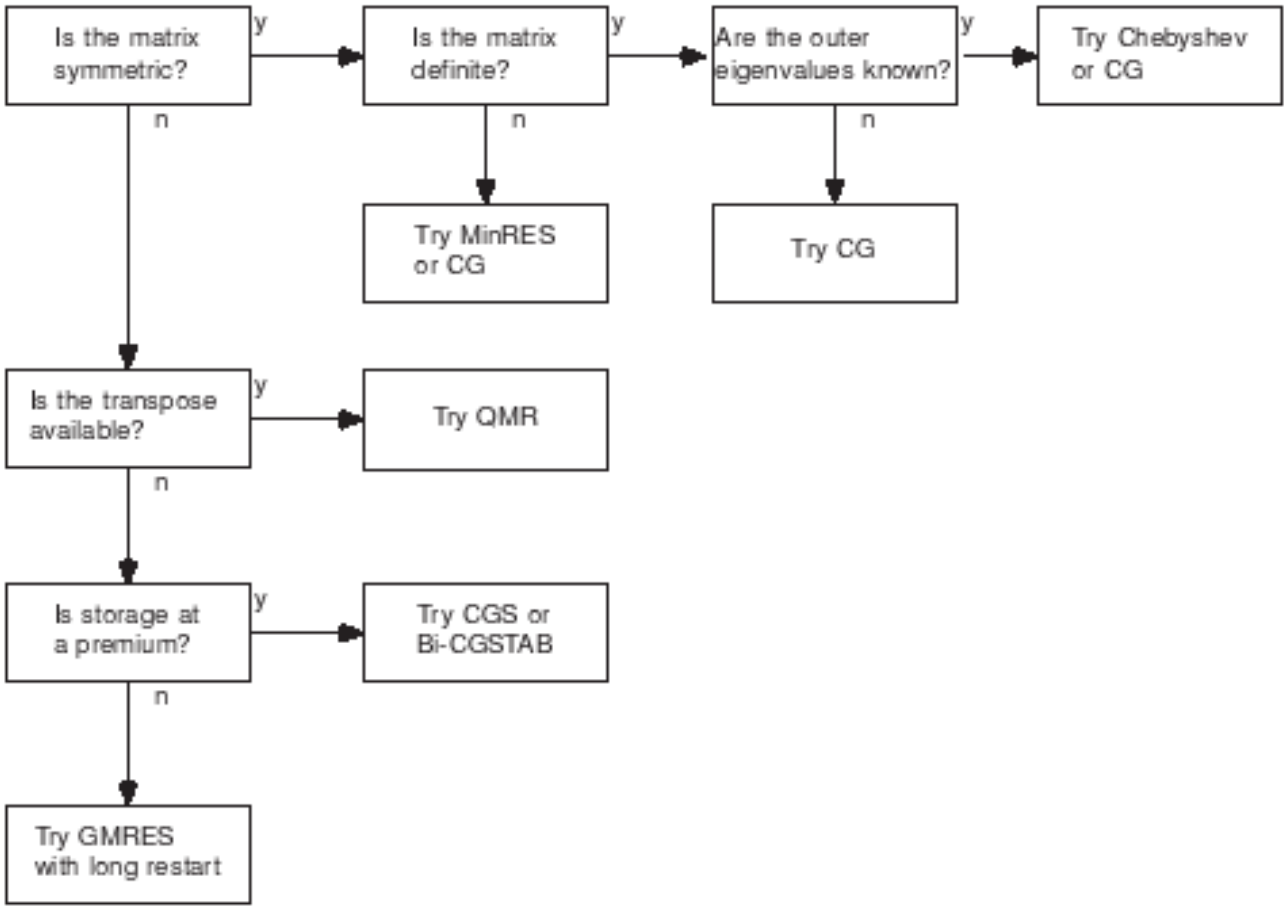
(worth spending a few minutes on this list...)
We will explore a few of these (but only a few) in greater detail.

Templates for Iterative Methods

- Originally published in 1994, 2nd edition available online
- R. Barrett et. al. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods* (SIAM, 2006)
- "Templates," in the sense of providing outlines for solutions of general class of problems using best-practice numerical methods
- Implementations of dense matrix code in `MATLAB` and `FORTRAN-77`
- Available on `netlib`: `www.netlib.org/templates`

Templates are somewhat older now, but an excellent code base nonetheless.

Flowchart for Templates/Iterative Methods



PETSc

PETSc - the Portable Extensible Toolkit Scientific Computation.

- Designed for ease of use (specifically for application developers)
- Parallel Data Structures (vectors, matrices, ghost points) and Solvers
- Scalable Parallel Preconditioners
- Several Sparse Storage Formats
- Automatic profiling features (floating point and memory)

PETSc Schematic

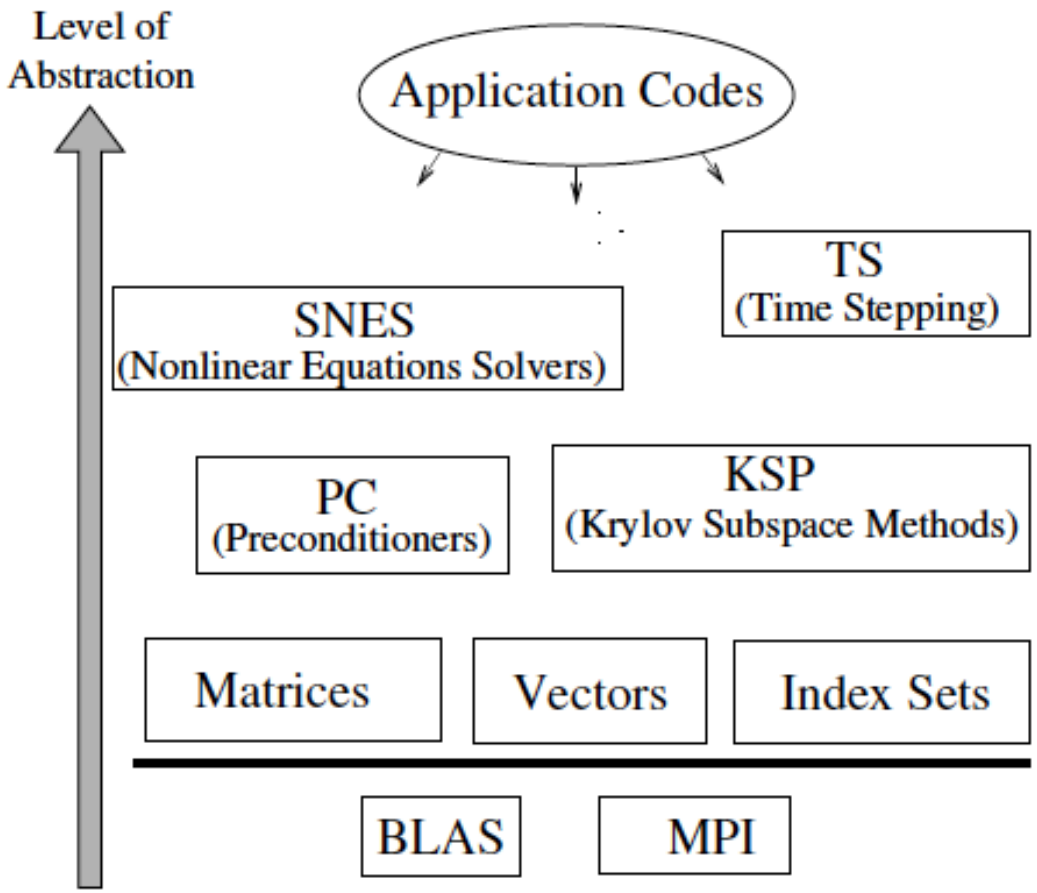


Figure 2: Organization of the PETSc Libraries

More PETSc

More PETSc Details:

- Applications can be written in `Fortran`, `C`, `C++`, or `Python`
- Large suite of linear and nonlinear equation solvers
- Extended via many add-on packages, e.g. SLEPC for sparse eigenvalue problems:
  `http://www.grycap.upv.es/slepc`

PETSc Numerical Libraries

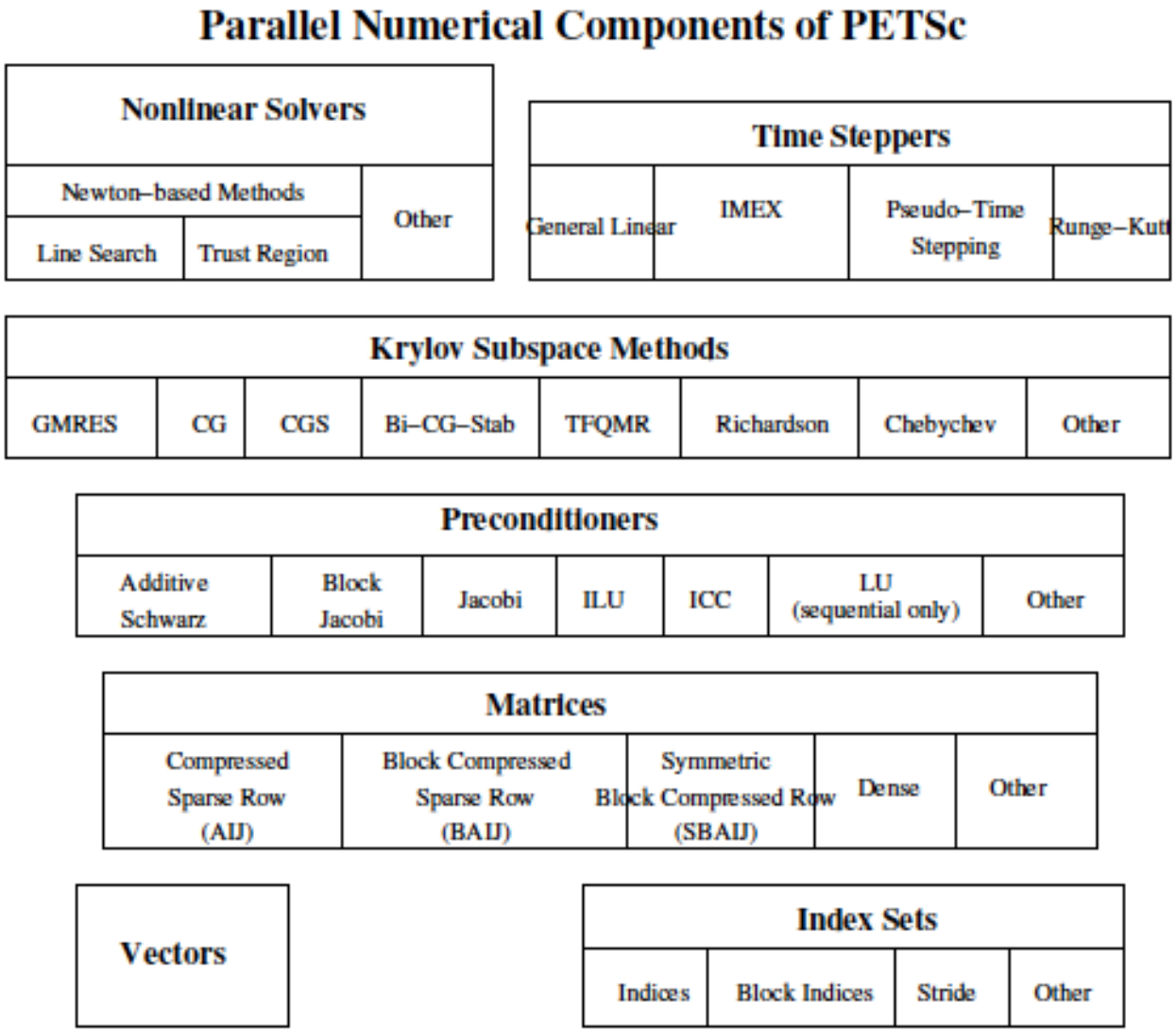


Figure 3: Numerical Libraries of PETSc

PETSc Simple Example

One of the "simple" PETSc examples, solving 2D Laplacian (discretized with finite differences):

```
1
2  /* Program usage:  mpiexec -n <procs> ex2 [-help] [all PETSc options] */
3
4  static char help[] = "Solves a linear system in parallel with KSP.\n\
5  Input parameters include:\n\
6    -random_exact_sol : use a random exact solution vector\n\
7    -view_exact_sol   : write exact solution vector to stdout\n\
8    -m <mesh_x>       : number of mesh points in x-direction\n\
9    -n <mesh_n>       : number of mesh points in y-direction\n\n";
10
11 /*T
12    Concepts: KSP^basic parallel example;
13    Concepts: KSP^Laplacian, 2d
14    Concepts: Laplacian, 2d
15    Processors: n
16 T*/
```

```
17
18  /*
19    Include "petscksp.h" so that we can use KSP solvers.  Note that this file
20    automatically includes:
21       petscsys.h       - base PETSc routines   petscvec.h - vectors
22       petscmat.h - matrices
23       petscis.h     - index sets            petscksp.h - Krylov subspace methods
24       petscviewer.h - viewers               petscpc.h  - preconditioners
25  */
26  #include "petscksp.h"
27
28  #undef __FUNCT__
29  #define __FUNCT__ "main"
30  int main(int argc,char **args)
31  {
32    Vec            x,b,u;  /* approx solution, RHS, exact solution */
33    Mat            A;        /* linear system matrix */
34    KSP            ksp;     /* linear solver context */
35    PetscRandom    rctx;     /* random number generator context */
36    PetscReal      norm;     /* norm of solution error */
37    PetscInt       i,j,Ii,J,Istart,Iend,m = 8,n = 7,its;
38    PetscErrorCode ierr;
39    PetscTruth     flg = PETSC_FALSE;
40    PetscScalar    v,one = 1.0,neg_one = -1.0;
41  #if defined(PETSC_USE_LOG)
42    PetscLogStage  stage;
43  #endif
```

PETSc Header

Typically one includes a single high-level PETSc header file that in turn includes lower-level header files (which you can directly include instead if you do not need the high-level functionality).

```
1  #include "petsc.h"      /* The whole enchilada */
2  /*
3
4  #include "petscksp.h"   /* Krylov subspace solvers */
5  #include "petscsnes.h"  /* Nonlinear solvers */
6  #include "petscts.h"    /* Time steppers */
7
8  */
```

The example just includes `petscksp.h` which in turn loads its lower-level dependent includes.

```
44
45    PetscInitialize(&argc,&args,(char *)0,help);
46    ierr = PetscOptionsGetInt(PETSC_NULL,"-m",&m,PETSC_NULL);CHKERRQ(ierr);
47    ierr = PetscOptionsGetInt(PETSC_NULL,"-n",&n,PETSC_NULL);CHKERRQ(ierr);
48    /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
49          Compute the matrix and right-hand-side vector that define
50          the linear system, Ax = b.
51       - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
52    /*
53       Create parallel matrix, specifying only its global dimensions.
54       When using MatCreate(), the matrix format can be specified at
55       runtime. Also, the parallel partitioning of the matrix is
56       determined by PETSc at runtime.
57
58       Performance tuning note:  For problems of substantial size,
59       preallocation of matrix memory is crucial for attaining good
60       performance. See the matrix chapter of the users manual for details.
61    */
62    ierr = MatCreate(PETSC_COMM_WORLD,&A);CHKERRQ(ierr);
63    ierr = MatSetSizes(A,PETSC_DECIDE,PETSC_DECIDE,m*n,m*n);CHKERRQ(ierr);
64    ierr = MatSetFromOptions(A);CHKERRQ(ierr);
65    ierr = MatMPIAIJSetPreallocation(A,5,PETSC_NULL,5,PETSC_NULL);CHKERRQ(ierr);
66    ierr = MatSeqAIJSetPreallocation(A,5,PETSC_NULL);CHKERRQ(ierr);
```

PETSc Data Layout

Data layout initialization routines:

- `MatCreate()` creates matrix
- `MatSetSizes()` sets local (`PETSC_DECIDE` leaves it up to PETSc) and global size
- `MatSetFromOptions()` creates matrix based on options passed on the command line, parallel if > 1 process, default type is `AIJ`, also known as the Yale sparse matrix or compressed row storage (nonzero elements by row with an array of corresponding column numbers and an array of pointers to beginning of each row).
- `MatMPIAIJSetPreallocation()` Pre-allocates memory for sparse parallel matrix, in this case a 5-point stencil is in use.
- `MatSeqAIJSetPreallocation()` Pre-allocates memory for local sparse matrix.

```
68  /*
69     Currently, all PETSc parallel matrix formats are partitioned by
70     contiguous chunks of rows across the processors.  Determine which
71     rows of the matrix are locally owned.
72  */
73  ierr = MatGetOwnershipRange(A,&Istart,&Iend);CHKERRQ(ierr);
74
75  /*
76     Set matrix elements for the 2-D, five-point stencil in parallel.
77      - Each processor needs to insert only elements that it owns
78        locally (but any non-local elements will be sent to the
79        appropriate processor during matrix assembly).
80      - Always specify global rows and columns of matrix entries.
81
82     Note: this uses the less common natural ordering that orders first
83     all the unknowns for x = h then for x = 2h etc; Hence you see J = Ii +- n
84     instead of J = I +- m as you might expect. The more standard ordering
85     would first do all variables for y = h, then y = 2h etc.
86
87   */
88  ierr = PetscLogStageRegister("Assembly", &stage);CHKERRQ(ierr);
89  ierr = PetscLogStagePush(stage);CHKERRQ(ierr);
90  for (Ii=Istart; Ii<Iend; Ii++) {
91    v = -1.0; i = Ii/n; j = Ii - i*n;
92    if (i>0)   {J=Ii-n; ierr=MatSetValues(A,1,&Ii,1,&J,&v,INSERT_VALUES);CHKERRQ(ierr);}
93    if (i<m-1) {J=Ii+n; ierr=MatSetValues(A,1,&Ii,1,&J,&v,INSERT_VALUES);CHKERRQ(ierr);}
94    if (j>0)   {J=Ii-1; ierr=MatSetValues(A,1,&Ii,1,&J,&v,INSERT_VALUES);CHKERRQ(ierr);}
95    if (j<n-1) {J=Ii+1; ierr=MatSetValues(A,1,&Ii,1,&J,&v,INSERT_VALUES);CHKERRQ(ierr);}
96    v = 4.0; ierr = MatSetValues(A,1,&Ii,1,&Ii,&v,INSERT_VALUES);CHKERRQ(ierr);
97  }
```

PETSc Data Initialization

Highlights:

- `PetscLogStageRegister()` custom logging/profiling (up to 10 stages) for use when using `-log_summary`
- `PetscLogStagePush()` commits to start custom logging/profiling
- `MatGetOwnershipRange()` range of rows owned by this parallel process
- `MetSetValues()` insert block of values into matrix, in this case the values are set by our 5-point stencil in 2D.

```
98    /*
99       Assemble matrix, using the 2-step process:
100        MatAssemblyBegin(), MatAssemblyEnd()
101      Computations can be done while messages are in transition
102      by placing code between these two statements.
103   */
104   ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
105   ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
106   ierr = PetscLogStagePop();CHKERRQ(ierr);
107
108   /*
109      Create parallel vectors.
110       - We form 1 vector from scratch and then duplicate as needed.
111       - When using VecCreate(), VecSetSizes and VecSetFromOptions()
112         in this example, we specify only the
113         vector's global dimension; the parallel partitioning is determined
114         at runtime.
115       - When solving a linear system, the vectors and matrices MUST
116         be partitioned accordingly.  PETSc automatically generates
117         appropriately partitioned matrices and vectors when MatCreate()
118         and VecCreate() are used with the same communicator.
119       - The user can alternatively specify the local vector and matrix
120         dimensions when more sophisticated partitioning is needed
121         (replacing the PETSC_DECIDE argument in the VecSetSizes() statement
122         below).
123   */
124   ierr = VecCreate(PETSC_COMM_WORLD,&u);CHKERRQ(ierr);
125   ierr = VecSetSizes(u,PETSC_DECIDE,m*n);CHKERRQ(ierr);
126   ierr = VecSetFromOptions(u);CHKERRQ(ierr);
127   ierr = VecDuplicate(u,&b);CHKERRQ(ierr);
128   ierr = VecDuplicate(b,&x);CHKERRQ(ierr);
```

PETSc Data Layout

Highlights:

- `MatAssemblyBegin()` begins assembling the parallel matrix (`MatSetValues` caches them for performance)
- `MatAssemblyEnd()` you can overlap more work while assembly is happening
- `PetscLogStagePop()` stop stage of custom logging/profiling
- `Vec*()` set up solution and right-hand side vectors

```
129   /*
130      Set exact solution; then compute right-hand-side vector.
131      By default we use an exact solution of a vector with all
132      elements of 1.0;  Alternatively, using the runtime option
133      -random_sol forms a solution vector with random components.
134   */
135   ierr=PetscOptionsGetTruth(PETSC_NULL,"-random_exact_sol",&flg,PETSC_NULL);CHKERRQ(ierr);
136   if (flg) {
137     ierr = PetscRandomCreate(PETSC_COMM_WORLD,&rctx);CHKERRQ(ierr);
138     ierr = PetscRandomSetFromOptions(rctx);CHKERRQ(ierr);
139     ierr = VecSetRandom(u,rctx);CHKERRQ(ierr);
140     ierr = PetscRandomDestroy(rctx);CHKERRQ(ierr);
141   } else {
142     ierr = VecSet(u,one);CHKERRQ(ierr);
143   }
144   ierr = MatMult(A,u,b);CHKERRQ(ierr);
145
146   /*
147      View the exact solution vector if desired
148   */
149   flg  = PETSC_FALSE;
150   ierr = PetscOptionsGetTruth(PETSC_NULL,"-view_exact_sol",&flg,PETSC_NULL);CHKERRQ(ierr);
151   if (flg) {ierr = VecView(u,PETSC_VIEWER_STDOUT_WORLD);CHKERRQ(ierr);}
```

```
152   /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
153                  Create the linear solver and set various options
154      - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
155
156   /*
157      Create linear solver context
158   */
159   ierr = KSPCreate(PETSC_COMM_WORLD,&ksp);CHKERRQ(ierr);
160
161   /*
162      Set operators. Here the matrix that defines the linear system
163      also serves as the preconditioning matrix.
164   */
165   ierr = KSPSetOperators(ksp,A,A,DIFFERENT_NONZERO_PATTERN);CHKERRQ(ierr);
166
167   /*
168      Set linear solver defaults for this problem (optional).
169      - By extracting the KSP and PC contexts from the KSP context,
170        we can then directly call any KSP and PC routines to set
171        various options.
172      - The following two statements are optional; all of these
173        parameters could alternatively be specified at runtime via
174        KSPSetFromOptions().  All of these defaults can be
175        overridden at runtime, as indicated below.
176   */
177   ierr = KSPSetTolerances(ksp,1.e-2/((m+1)*(n+1)),1.e-50,PETSC_DEFAULT,
178                           PETSC_DEFAULT);CHKERRQ(ierr);
```

```
179   /*
180     Set runtime options, e.g.,
181         -ksp_type <type> -pc_type <type> -ksp_monitor -ksp_rtol <rtol>
182     These options will override those specified above as long as
183     KSPSetFromOptions() is called _after_ any other customization
184     routines.
185   */
186   ierr = KSPSetFromOptions(ksp);CHKERRQ(ierr);
187
188   /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
189                       Solve the linear system
190      - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
191
192   ierr = KSPSolve(ksp,b,x);CHKERRQ(ierr);
```

```
193   /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
194                       Check solution and clean up
195      - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
196   /*
197      Check the error
198   */
199   ierr = VecAXPY(x,neg_one,u);CHKERRQ(ierr);
200   ierr = VecNorm(x,NORM_2,&norm);CHKERRQ(ierr);
201   ierr = KSPGetIterationNumber(ksp,&its);CHKERRQ(ierr);
202   /* Scale the norm */
203   /*  norm *= sqrt(1.0/((m+1)*(n+1))); */
204   /*
205      Print convergence information.  PetscPrintf() produces a single
206      print statement from all processes that share a communicator.
207      An alternative is PetscFPrintf(), which prints to a file.
208   */
209   ierr = PetscPrintf(PETSC_COMM_WORLD,"Norm of error %A iterations %D\n",
210                      norm,its);CHKERRQ(ierr);
211   /*
212      Free work space.  All PETSc objects should be destroyed when they
213      are no longer needed.
214   */
215   ierr = KSPDestroy(ksp);CHKERRQ(ierr);
216   ierr = VecDestroy(u);CHKERRQ(ierr);  ierr = VecDestroy(x);CHKERRQ(ierr);
217   ierr = VecDestroy(b);CHKERRQ(ierr);  ierr = MatDestroy(A);CHKERRQ(ierr);
218   /*
219      Always call PetscFinalize() before exiting a program.  This routine
220        - finalizes the PETSc libraries as well as MPI
221        - provides summary and diagnostic information if certain runtime
222          options are chosen (e.g., -log_summary).   */
223   ierr = PetscFinalize();CHKERRQ(ierr);
224   return 0;
225 }
```

PETSc Makefiles

PETSc has an extensive Makefile system that it uses to facilitate code management and build options:

- The directory `$PETSC_DIR/conf` contains Makefile includes and customizations (architecture dependent ones are in `$PETSC_DIR/$PETSC_ARCH/conf`)
- `$PETSC_LIB` is a short variable that will link in the entire PETSc set of libraries in the right order, you can select a subset using other environment variables: `$PETSC_SYS_LIB`, `$PETSC_VEC_LIB`, `$PETSC_MAT_LIB`, `$PETSC_DM_LIB`, `$PETSC_KSP_LIB`, `$PETSC_SNES_LIB`, `$PETSC_TS_LIB`

PETSc Makefile Example

From the PETSc examples for linear solvers:

```
CFLAGS           =
FFLAGS           =
CPPFLAGS         =
FPPFLAGS         =

include ${PETSC_DIR}/conf/variables
include ${PETSC_DIR}/conf/rules

ex2: ex2.o
        ${CLINKER} -o ex2 ex2.o  ${PETSC_KSP_LIB}
        ${RM} ex2.o
```

PETSc Example (cont'd)

```
[rush:~/d_petsc/ex2]$ module
Currently Loaded Modulefiles:
  1) null                          4) intel-mpi/4.1.1       7) valgrind/3.8.1-impi-4.1.1
  2) modules                       5) intel/13.1            8) petsc/v3.4.2-mpi
  3) use.own                       6) mkl/11.1
[rush:~/d_petsc/ex2]$ make ex2
/util/intel/impi/4.1.1.036/intel64/bin/mpiicc -o ex2.o -c -fPIC -wd1572 -g \
-I/util/petsc/petsc-3.4.2/include -I/util/petsc/petsc-3.4.2/linux-impi-mkl/include \
-I/util/perftools/papi-5.1.1/include -I/util/valgrind/3.8.1/include \
-I/util/intel/impi/4.1.1.036/intel64/include    \
-D__INSDIR__=src/ksp/ksp/examples/tutorials/ ex2.c
/util/intel/impi/4.1.1.036/intel64/bin/mpiicc -fPIC -wd1572 -g -o ex2 ex2.o \
 -Wl,-rpath,/util/petsc/petsc-3.4.2/linux-impi-mkl/lib \
 -L/util/petsc/petsc-3.4.2/linux-impi-mkl/lib  -lpetsc \
 -Wl,-rpath,/util/petsc/petsc-3.4.2/linux-impi-mkl/lib -lHYPRE \
 -Wl,-rpath,/util/intel/impi/4.1.1.036/intel64/lib \
 -L/util/intel/impi/4.1.1.036/intel64/lib \
 -Wl,-rpath,/ifs/util/util64/intel/composer_xe_2013.5.192/compiler/lib/intel64 \
 -L/ifs/util/util64/intel/composer_xe_2013.5.192/compiler/lib/intel64 \
 -Wl,-rpath,/usr/lib/gcc/x86_64-redhat-linux/4.4.7 \
 -L/usr/lib/gcc/x86_64-redhat-linux/4.4.7 \
 -Wl,-rpath,/ifs/util/util64/petsc/petsc-3.4.2/-Xlinker -lmpigc4 \
 -Wl,-rpath,/opt/intel/mpi-rt/4.1 -Wl,-rpath,/util/intel/composer_xe_2013/mkl/lib/intel64 \
 -L/util/intel/composer_xe_2013/mkl/lib/intel64 \
 -lmkl_intel_lp64 -lmkl_sequential -lmkl_core -lpthread -lm -lX11 -lpthread \
 -Wl,-rpath,/util/perftools/papi-5.1.1/lib -L/util/perftools/papi-5.1.1/lib \
 -lpapi -lmpi -lifport -lifcore -lm -lm -lmpigc4 -ldl -lmpigf -lmpi_dbg -lmpigi \
 -lrt -lpthread -limf -lsvml -lirng -lipgo -ldecimal -lcilkrts -lstdc++ -lgcc_s \
 -lirc -lirc_s -ldl
/bin/rm -f ex2.o
```

PETSc Example (cont'd)

There are too many run-time options to PETSc to conveniently list (try `-help` when you run, be prepared for a lot of options). For a Krylov subspace solver example, `-log_summary`, `-ksp_monitor`, and `-ksp_type cg` are handy options to explore. You can also set the grid size using `-m` and `-n`, and the tolerance with `-ksp_rtol`.

```
1 [rush:~/d_petsc/ex2]$ mpirun -np 4 ./ex2 -m 256 -n 256
2 Norm of error 0X1.40B8691A846F4P-8 iterations 443
3 [rush:~/d_petsc/ex2]$ mpirun -np 4 ./ex2 -m 256 -n 256 -ksp_rtol 1.e-8
4 Norm of error 0X1.7CE3C77A40768P-12 iterations 524
5 [rush:~/d_petsc/ex2]$ mpirun -np 4 ./ex2 -m 256 -n 256 -ksp_rtol 1.e-10
6 Norm of error 0X1.5FD64CC703D6EP-19 iterations 692
```

-log_summary

```
[rush:~/d_petsc/ex2]$ mpirun -np 8 ./ex2 -m 512 -n 512 -ksp_rtol 1.e-10 -log_summary
Norm of error 0X1.8748AE369060CP-16 iterations 3014
************************************************************************************************************************
***             WIDEN YOUR WINDOW TO 120 CHARACTERS.  Use 'enscript -r -fCourier9' to print this document            ***
************************************************************************************************************************

---------------------------------------------- PETSc Performance Summary: ----------------------------------------------

./ex2 on a linux-impi-mkl named f07n05 with 8 processors, by jonesm Mon Oct 21 15:07:07 2013
Using Petsc Release Version 3.4.2, Jul, 02, 2013

                         Max       Max/Min        Avg      Total
Time (sec):           5.011e+01      1.00000   5.011e+01
Objects:              5.600e+01      1.00000   5.600e+01
Flops:                8.455e+09      1.00038   8.454e+09  6.763e+10
Flops/sec:            1.687e+08      1.00038   1.687e+08  1.350e+09
Memory:               1.685e+07      1.00110              1.348e+08
MPI Messages:         6.234e+03      2.00000   5.455e+03  4.364e+04
MPI Message Lengths:  2.552e+07      2.00000   4.094e+03  1.787e+08
MPI Reductions:       6.236e+04      1.00000

Flop counting convention: 1 flop = 1 real number operation of type (multiply/divide/add/subtract)
                            e.g., VecAXPY() for real vectors of length N --> 2N flops
                            and VecAXPY() for complex vectors of length N --> 8N flops

Summary of Stages:   ----- Time ------  ----- Flops -----  --- Messages ---  -- Message Lengths --  -- Reductions --
                        Avg     %Total     Avg     %Total   counts   %Total     Avg         %Total   counts   %Total
 0:      Main Stage: 4.9592e+01  99.0%  6.7633e+10 100.0%  4.361e+04  99.9%  4.093e+03      100.0%  6.233e+04 100.0%
 1:        Assembly: 5.1608e-01   1.0%  0.0000e+00   0.0%  2.800e+01   0.1%  6.583e-01        0.0%  2.300e+01   0.0%

------------------------------------------------------------------------------------------------------------------------
```

```
See the 'Profiling' chapter of the users' manual for details on interpreting output.
Phase summary info:
   Count: number of times phase was executed
   Time and Flops: Max - maximum over all processors
                   Ratio - ratio of maximum to minimum over all processors
   Mess: number of messages sent
   Avg. len: average message length (bytes)
   Reduct: number of global reductions
   Global: entire computation
   Stage: stages of a computation. Set stages with PetscLogStagePush() and PetscLogStagePop().
      %T - percent time in this phase         %f - percent flops in this phase
      %M - percent messages in this phase     %L - percent message lengths in this phase
      %R - percent reductions in this phase
   Total Mflop/s: 10e-6 * (sum of flops over all processors)/(max time over all processors)
------------------------------------------------------------------------------------------------------------------------
      ##########################################################
      #                                                        #
      #                          WARNING!!!                    #
      #                                                        #
      #   This code was compiled with a debugging option,      #
      #   To get timing results run ./configure                #
      #   using --with-debugging=no, the performance will      #
      #   be generally two or three times faster.              #
      #                                                        #
      ##########################################################
Event                Count      Time (sec)     Flops                             --- Global ---  --- Stage ---   Total
                   Max Ratio  Max     Ratio   Max  Ratio  Mess   Avg len Reduct  %T %f %M %L %R  %T %f %M %L %R Mflop/s
------------------------------------------------------------------------------------------------------------------------
--- Event Stage 0: Main Stage
ThreadCommRunKer       1 1.0 2.1935e-05 1.4 2.40e+01 1.1 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0      8
ThreadCommBarrie       1 1.0 8.8215e-06 1.1 2.60e+01 1.2 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     20
MatMult             3115 1.0 7.0171e+00 1.0 1.02e+09 1.0 4.4e+04 4.1e+03 0.0e+00 14 12100100  0  14 12100100  0  1164
MatSolve            3115 1.0 6.3962e+00 1.0 9.13e+08 1.0 0.0e+00 0.0e+00 0.0e+00 13 11  0  0  0  13 11  0  0  0  1142
MatCholFctrNum         1 1.0 1.8241e-02 1.1 1.07e+06 1.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0   469
MatICCFactorSym        1 1.0 7.3059e-03 1.0 3.40e+01 1.1 0.0e+00 0.0e+00 1.0e+00  0  0  0  0  0   0  0  0  0  0     0
MatGetRowIJ            1 1.0 2.8849e-05 2.6 2.30e+01 1.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     6
MatGetOrdering         1 1.0 6.2199e-03 1.0 7.80e+01 1.1 0.0e+00 0.0e+00 2.0e+00  0  0  0  0  0   0  0  0  0  0     0
VecMDot             3014 1.0 6.7773e+00 1.5 3.06e+09 1.0 0.0e+00 0.0e+00 3.0e+03 10 36  0  0  5  11 36  0  0  5  3612
VecNorm             3116 1.0 3.0052e-01 1.2 1.21e+08 1.0 0.0e+00 0.0e+00 3.1e+03  1  1  0  0  5   1  1  0  0  5  3226
VecScale            3115 1.0 3.2919e-01 1.1 5.11e+07 1.0 0.0e+00 0.0e+00 0.0e+00  1  1  0  0  0   1  1  0  0  0  1243
VecCopy              101 1.0 1.3905e-02 1.4 2.35e+03 1.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     1
VecSet              3218 1.0 2.6570e-01 1.3 7.39e+04 1.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   1  0  0  0  0     2
VecAXPY              202 1.0 2.9503e-02 1.2 6.69e+06 1.0 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0  1813
VecMAXPY            3115 1.0 5.5059e+00 1.0 3.27e+09 1.0 0.0e+00 0.0e+00 0.0e+00 11 39  0  0  0  11 39  0  0  0  4757
VecScatterBegin     3115 1.0 8.1264e-02 1.3 1.06e+05 1.2 4.4e+04 4.1e+03 0.0e+00  0  0100100  0   0  0100100  0    10
VecScatterEnd       3115 1.0 9.4404e-02 1.3 9.98e+04 1.1 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     8
VecNormalize        3115 1.0 7.1746e-01 1.1 1.73e+08 1.0 0.0e+00 0.0e+00 3.1e+03  1  2  0  0  5   1  2  0  0  5  1923
KSPGMRESOrthog      3014 1.0 1.2234e+01 1.2 6.14e+09 1.0 0.0e+00 0.0e+00 5.0e+04 21 73  0  0 80  22 73  0  0 80  4012
KSPSetUp               2 1.0 2.4221e-03 1.0 6.60e+01 1.1 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     0
KSPSolve               1 1.0 4.9552e+01 1.0 9.50e+09 1.0 4.4e+04 4.1e+03 6.2e+04 99112100100100 100112100100100  1528
PCSetUp                2 1.0 3.2593e-02 1.0 1.07e+06 1.0 0.0e+00 0.0e+00 1.8e+01  0  0  0  0  0   0  0  0  0  0   263
PCSetUpOnBlocks        1 1.0 3.1875e-02 1.0 1.07e+06 1.0 0.0e+00 0.0e+00 1.0e+01  0  0  0  0  0   0  0  0  0  0   269
PCApply             3115 1.0 1.3323e+01 1.1 1.18e+09 1.0 0.0e+00 0.0e+00 6.2e+03 26 14  0  0 10  26 14  0  0 10   704
--- Event Stage 1: Assembly
MatAssemblyBegin       1 1.0 2.0136e-02124.4 2.60e+01 1.0 0.0e+00 0.0e+00 2.0e+00  0  0  0  0  0   3  0  0  0  9     0
MatAssemblyEnd         1 1.0 1.2871e-02 1.0 1.79e+02 1.1 2.8e+01 1.0e+03 2.1e+01  0  0  0  0  0   2  0100100 91     0
VecSet                 1 1.0 3.9101e-05 1.5 2.40e+01 1.1 0.0e+00 0.0e+00 0.0e+00  0  0  0  0  0   0  0  0  0  0     5
```

```
Memory usage is given in bytes:

Object Type          Creations   Destructions    Memory  Descendants' Mem.
Reports information only for process 0.

--- Event Stage 0: Main Stage

              Matrix     4              4      6165988     0
              Vector    39             40      9764992     0
      Vector Scatter     0              1         1052     0
           Index Set     3              3       133352     0
       Krylov Solver     2              2        19504     0
      Preconditioner     2              2         1800     0
              Viewer     1              0            0     0

--- Event Stage 1: Assembly

              Vector     2              1         1544     0
      Vector Scatter     1              0            0     0
           Index Set     2              2         3560     0
========================================================================================================================
Average time to get PetscTime(): 0
Average time for MPI_Barrier(): 3.38554e-06
Average time for zero size MPI_Send(): 1.0401e-05
#PETSc Option Table entries:
-ksp_rtol 1.e-10
-log_summary
-m 512
-n 512
#End of PETSc Option Table entries
Compiled without FORTRAN kernels
...
```

PETSc Summary

PETSc is flexible, extensive feature set:

- Designed for multiple discretization schemes
- Multiple solvers (even choosing at run-time)
- Parallelism in MPI (and now GPU)
- Leverages best available libraries and numerical support
- Learning curve is relatively steep - read the manual/tutorials before starting:

  http://www.mcs.anl.gov/petsc/petsc-as/documentation/index.html

SuperLU

SuperLU:

- Direct solution of large sparse non-symmetric matrices
- LU decomposition with partial pivoting and triangular system solves through forward and back substitution
- Iterative refinement subroutines provided for improved backward stability
- Routines also provided to equilibrate, estimate condition number, calculate relative backward error, and estimate error bounds for refined solutions
- Distributed memory version (MPI) available
- Also available through PETSc
- X. Li, "An overview of SuperLU: Algorithms, implementation, and user interface," ACM Trans. Math. Soft. (TOMS), **31**, 302-325 (2005).

Trilinos

Trilinos is from Greek for a string of pearls - actually three pearls, but Trilinos has long since evolved beyond its initial set of three packages:

- Object oriented framework (C++) for the solution of large-scale, complex multi-physics engineering and scientific problems
- Uses packages for supporting broad range of features and capabilities, each package is an independent piece of software with its own requirements, development team, and users
- http://trilinos.sandia.gov
- Similar to **petsc**, also built for distributed memory, leverage BLAS/LAPACK, etc., somewhat broader applicability and more diverse package system

Adaptive Mesh Refinement (AMR)

Adaptive Mesh Refinement (AMR):

- Requires careful handling of mesh data structures (picture ...)
- Load balancing can be difficult
- Packages available that help address both issues by providing library routines and data structures

SAMRAI

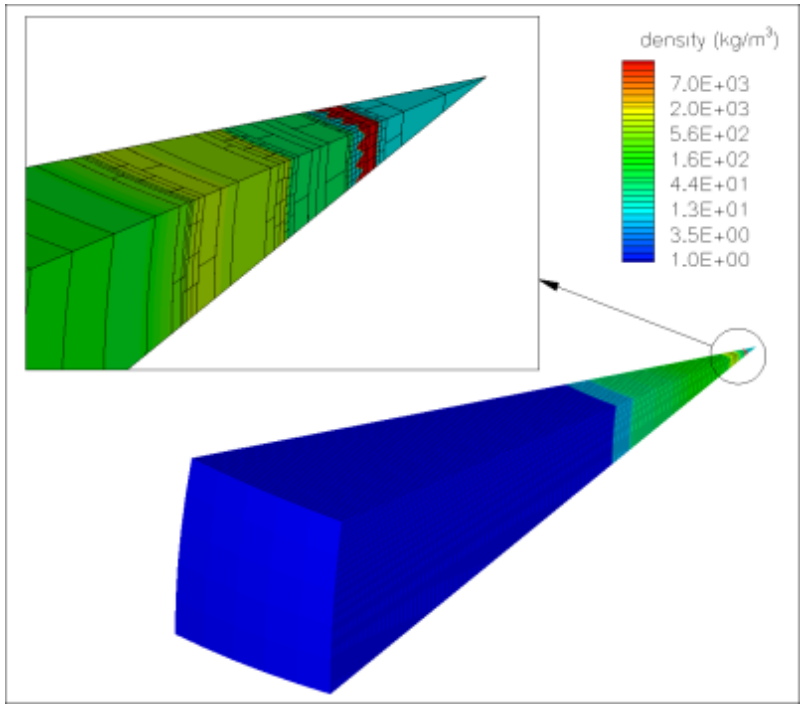
`SAMRAI`: Structured Adaptive Mesh Refinement Application Infrastructure

https://computation.llnl.gov/casc/SAMRAI

- Object-oriented C++ library
- User-controlled mesh refinement
- Customized adaptive meshing in arbitrary dimension
- Interface to solver libraries, including hypre, PeTSC, and SUNDIALS
- Built-in support for profiling and gathering statistics
- Support for flexible parallel I/O through HDF5

SAMRAI (cont'd)

Primarily developed for so-called multiphysics applications that couple together multiple length and/or time scales, and multiple areas of physical behavior (e.g., hydrodynamics and chemistry).



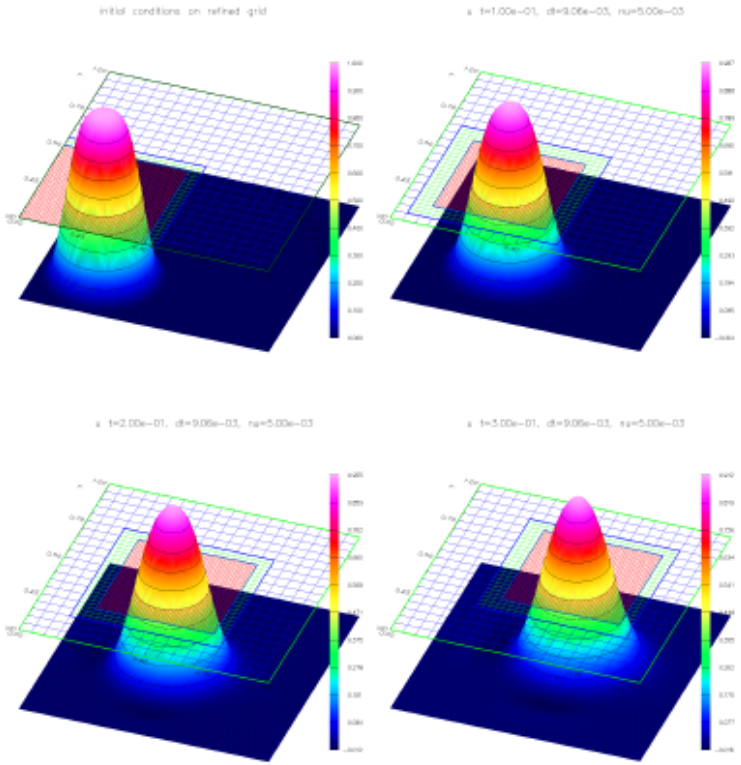
Inertial confinement fusion target simulation, spatial scale spans microns to centimeters, coupling Adaptive Lagrangian-Eulerian methods with AMR (ALE-AMR project).

Overture

`Overture`: `http://www.llnl.gov/casc/Overture/`

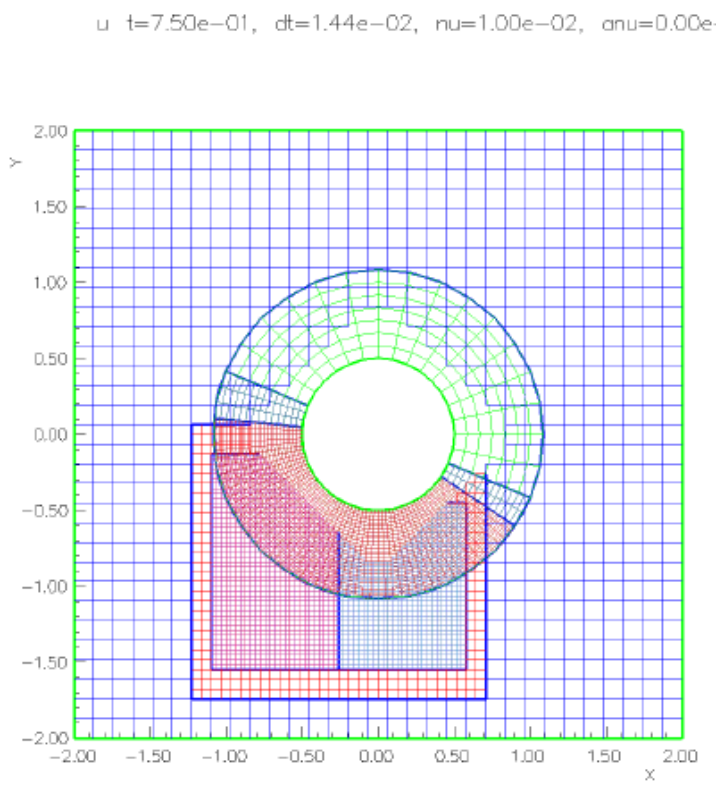
- Framework for solving PDEs in serial and parallel
- Object-oriented collection of libraries in `C++`
- Finite differences using collection of structured (and unstructured) grids
- Composite overlapping grids, curvilinear coordinates, adaptive mesh refinement
- Complex moving domains
- Built-in solvers, including PeTSC, SuperLU, multigrid capabilities
- Includes grid generation capability

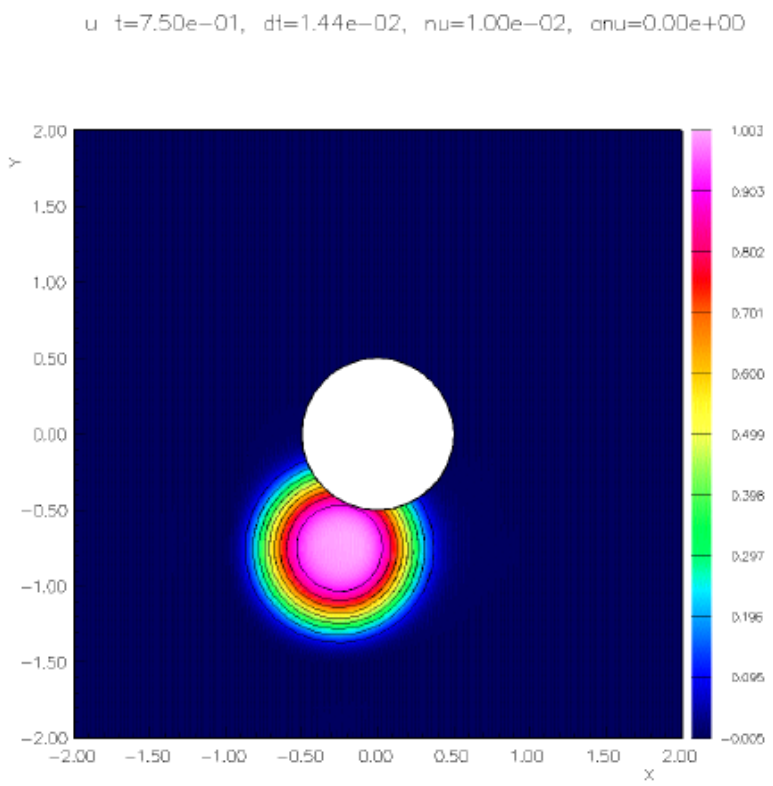
Overture (cont'd)



Convection diffusion equation example in `Overture` with AMR.

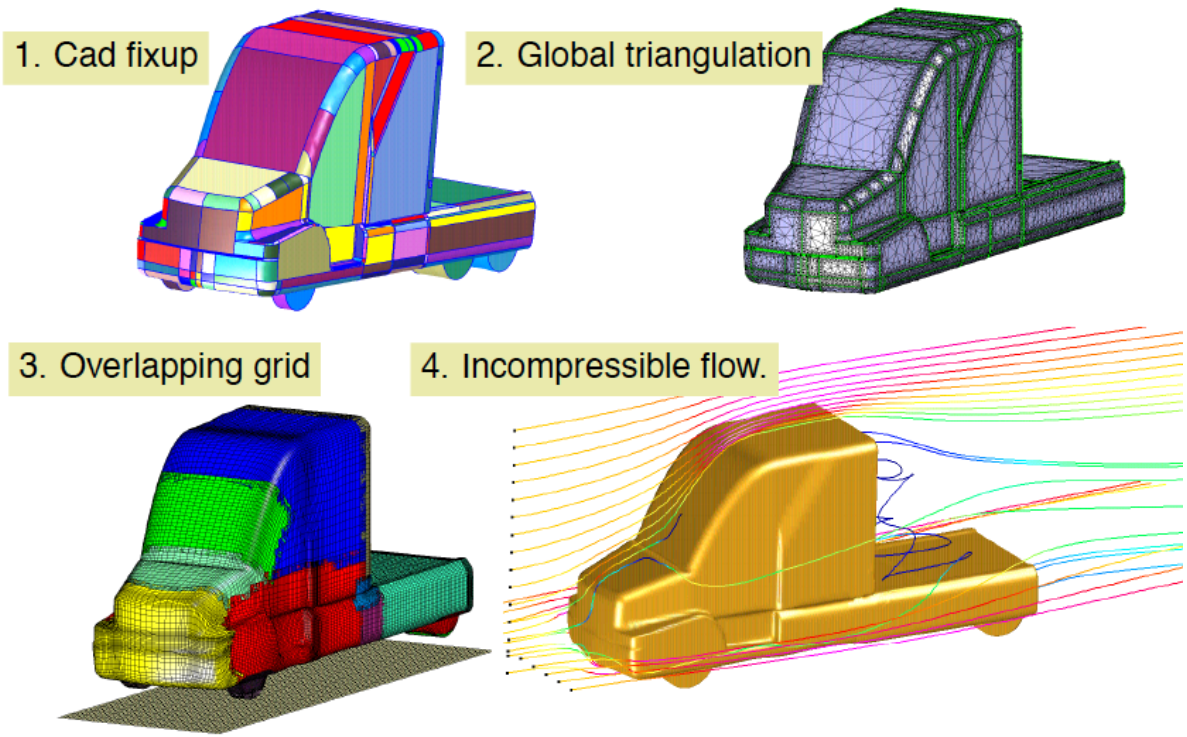
Overture (cont'd)





Convection diffusion equation, pulse crossing circle inside square.

Overture (cont'd)



Overture from CAD to flow.

Summary of Available Algorithms

Source: J. Demmel, *Applied Numerical Linear Algebra* (SIAM, 1997) dimension $N = n^2$ (2D), $N = n^3$ (3D). PRAM is an idealized zero-cost communication parallel model.

Algorithm	Serial	PRAM	Memory	Procs
Dense LU	N^3	N	N^2	N^2
Band LU	$N^2 (N^{7/3})$	N	$N^{3/2} (N^{5/3})$	N $(N^{4/3})$
Jacobi	$N^2 (N^{5/3})$	N	N	N
Explic. Inv.	N^2	$\log N$	N^2	N^2
Conj. Grad.	$N^{3/2} (N^{4/3})$	$\sqrt{N} \log N (N^{1/3} \log N)$	N	N
Red/Black SOR	$N^{3/2} (N^{4/3})$	$N^{1/2} (N^{1/3})$	N	N
Sparse LU	$N^{3/2} (N^2)$	$\sqrt{N}$	$N \log N$ $(N^{4/3})$	N
FFT	$N \log N$	$\log N$	N	N
Multigrid	N	$\log^2 N$	N	N

Algorithm Overview (cont'd)

- Dense LU: works on any matrix
- Band LU: Exploit nonzero only on sub-diagonals numbering $\sqrt{N}(N^{2/3})$.
- Jacobi: Matrix-vector multiplication
- Explicit Inverse: leverage many right-hand sides by pre-computing (and storing) inverse of left-hand side matrix
- Conjugate Gradient: Matrix-vector multiply like Jacobi, but uses mathematical properties of left-hand operator
- Red-Black SOR: Matrix-vector multiply like Jacobi, using updated solution
- Sparse LU: Gaussian elimination using sparsity of operator matrix
- FFT: restricted applicability

Algorithm Overview (cont'd)

- Dimension: $N = n^2$ (2D), $N = n^3$ (3D)
- Bandwidth: n (2D), n^2 (3D)
- Condition Number: $k = n^2$ (2D/3D)
- Dense LU: cost $\sim N^3$ (serial), $\sim N$ (parallel over N^2 processors
- Band LU: cost $\sim N \times (BW)^2$ (serial), $\sim N$ (parallel over (BW) processors
- Jacobi: cost is $\sim$ sparse matrix-vector multiply, number of iterations $\sim k$

Algorithm Overview (cont'd)

- Conjugate Gradient: cost $\sim$ (sparse matrix-vector multiply)(dot product)(number iterations), number of iterations $\sim \sqrt{k}$.
- Red-Black SOR: cost $\sim$ (sparse matrix-vector multiply)(number iterations), number of iterations $\sim \sqrt{k}$.