Lawrence Livermore National Laboratory

HYPRE: High Performance Preconditioners

USQCD Algorithms and Computing Workshop
Livermore Valley Open Campus, November 10 - 11, 2011

Robert D. Falgout

Center for Applied Scientific Computing

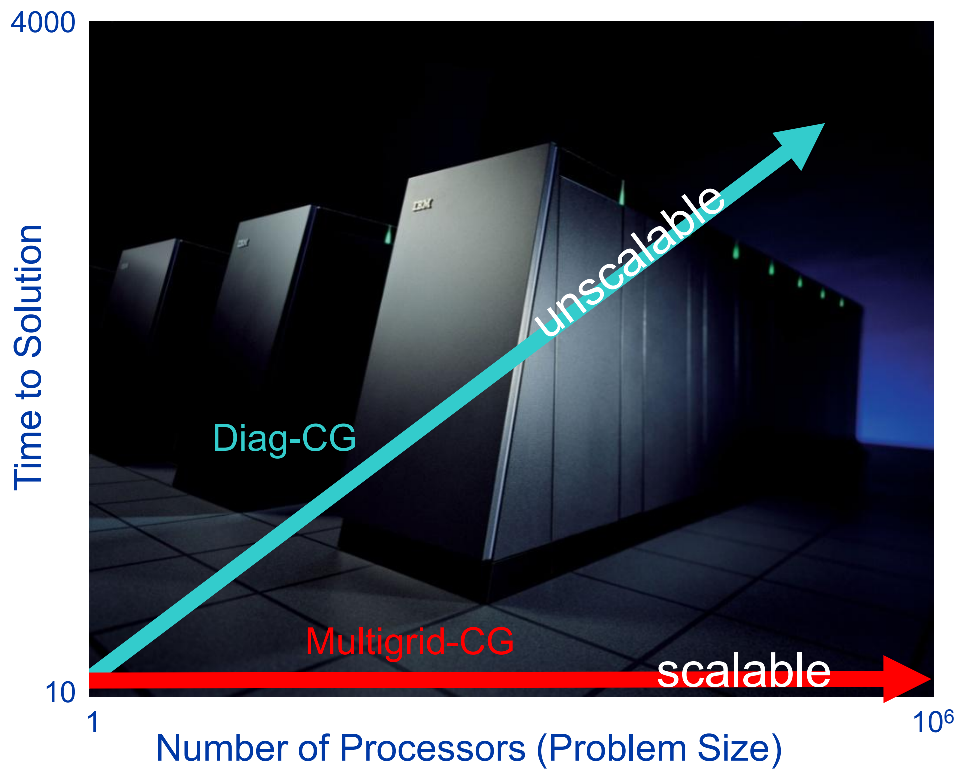
This work performed under the auspices of the U.S. Department of Energy by
Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344

LLNL-PRES-231999

Outline

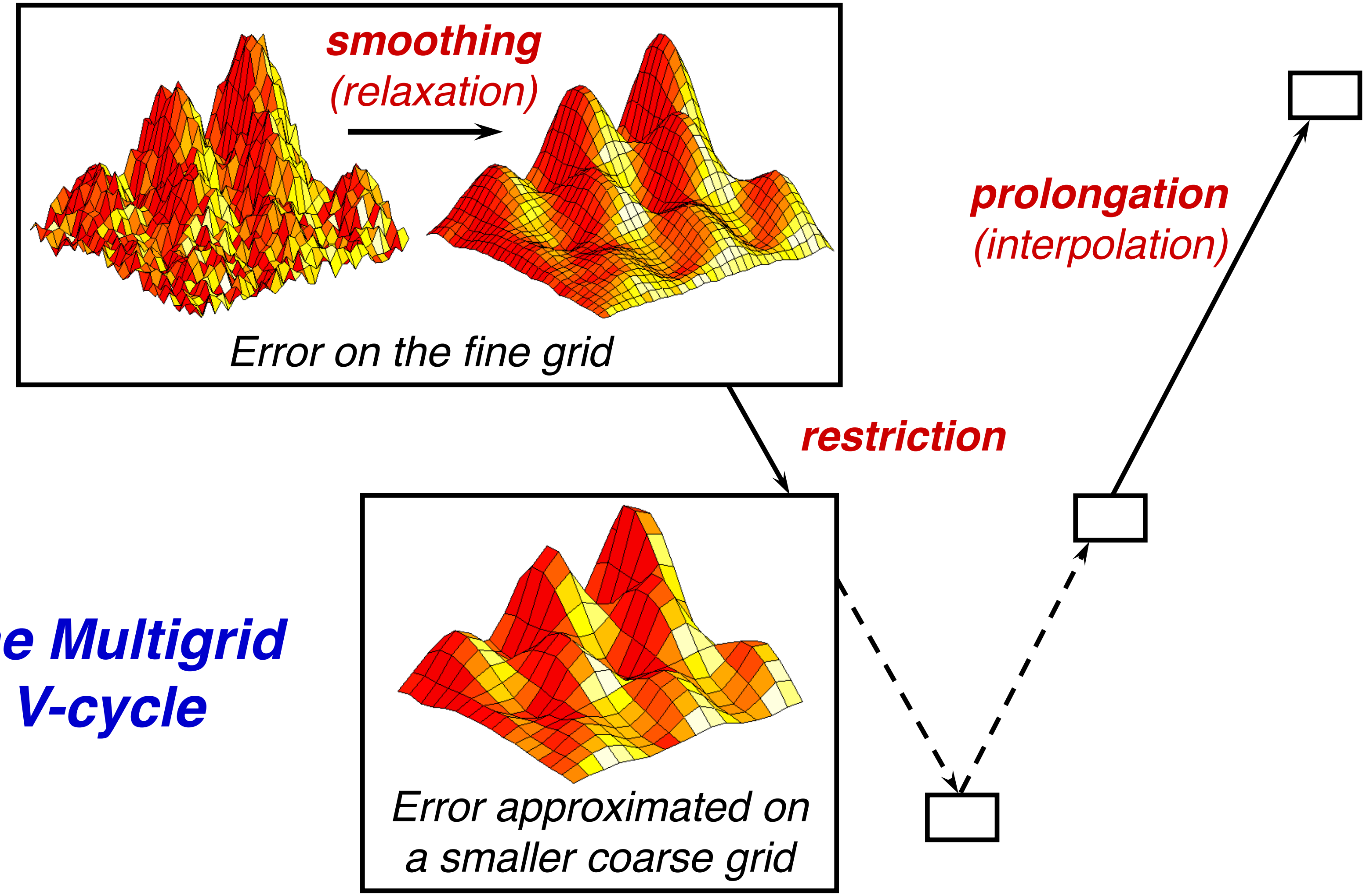
- Background
- Parallel Multigrid (towards exascale)
- HYPRE
- USQCD / FASTMath Interactions

Multigrid linear solvers are optimal ($O(N)$ operations), and hence have good scaling potential



- Weak scaling – want constant solution time as problem size grows in proportion to the number of processors

Multigrid uses a sequence of coarse grids to accelerate the fine grid solution



The Multigrid V-cycle

The basic multigrid research challenge

- Optimal O(N) multigrid methods don't exist for some applications, even in serial
- Need to invent methods for these applications

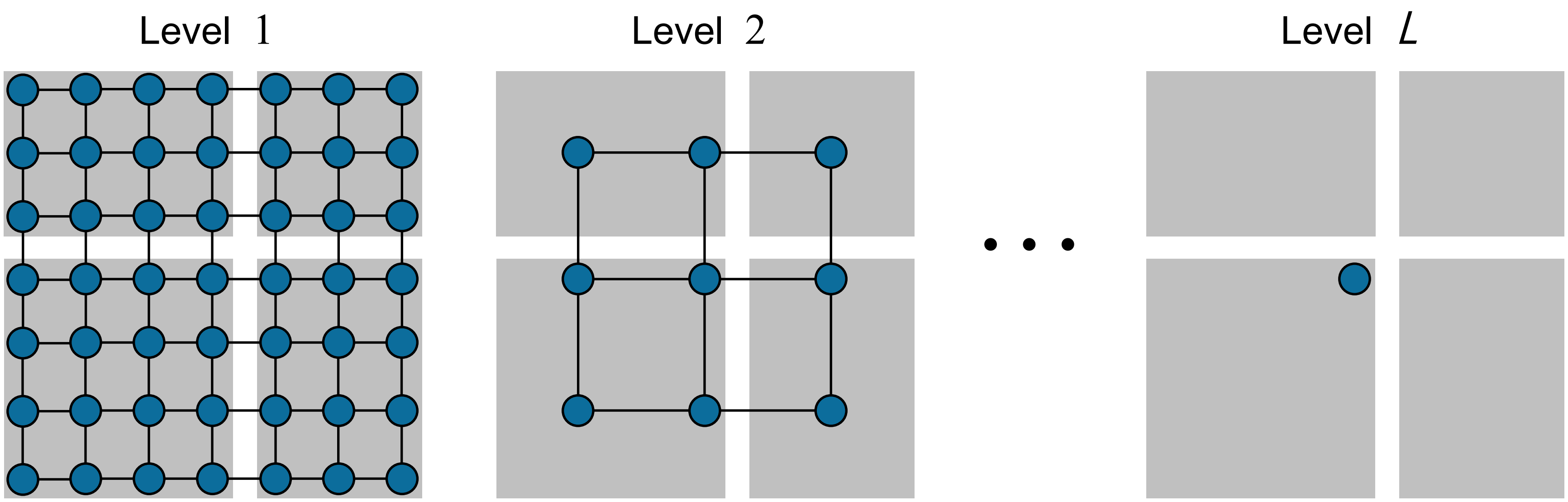
- However …

- Some of the classical and most proven techniques used in multigrid methods don't parallelize
  - Gauss-Seidel smoothers are inherently sequential
  - W-cycles have poor parallel scaling
- Parallel computing imposes additional restrictions on multigrid algorithmic development

- Tomorrow's exascale computers with huge core counts and small memories just magnifies the challenge

Parallel Multigrid

Approach for parallelizing multigrid is straightforward data decomposition



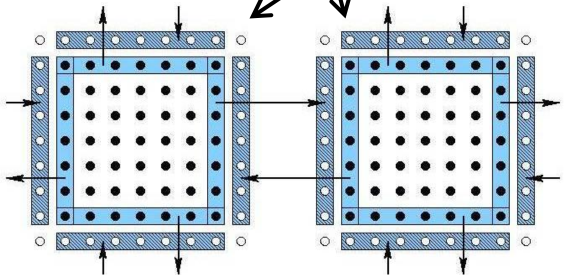
- Basic communication pattern is "nearest neighbor"
  - Relaxation, interpolation, & Galerkin not hard to implement
- Different neighbor processors on coarse grids
- Many idle processors on coarse grids (100K+ on BG/L)
  - Algorithms to take advantage have had limited success

Straightforward parallelization approach is optimal for V-cycles on structured grids (5-pt Laplacian example)

- Standard communication / computation models

$$T_{comm} = \alpha + m\beta \quad \textit{(communicate m doubles)}$$

$$T_{comp} = m\gamma \quad \textit{(compute m flops)}$$



- Time to do relaxation

$$T \approx 4\alpha + 4n\beta + 5n^2\gamma$$

- Time to do relaxation in a V(1,0) multigrid cycle

$$T_V \approx (1 + 1 + \cdots)4\alpha + (1 + 1/2 + \ldots)4n\beta + (1 + 1/4 + \ldots)5n^2\gamma$$
$$\approx (\log N)4\alpha + (2)4n\beta + (4/3)5n^2\gamma$$

- For achieving optimality in general, the *log* term is unavoidable!

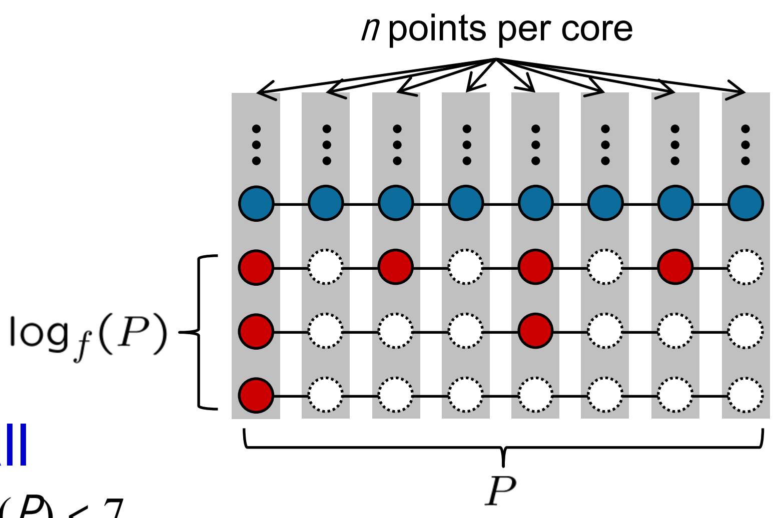
- More precise: $T_{V,better} \approx T_V + (\log P)(4\beta + 5\gamma)$

Idle processors (cores) are not really a problem

- The idle processor problem seems severe, but standard parallel V-cycle multigrid performance has optimal order
- What are the limits of what we can achieve by trying to use idle processors to accelerate convergence?

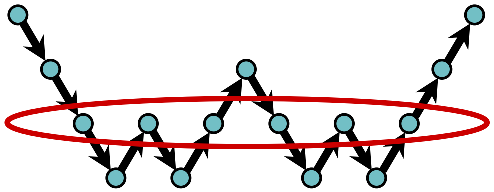
- Best overall speedup assuming at least one V-cycle is required

$$\max\left\{1 + \frac{\log_f(P)}{n}\,,\ \frac{C}{F}\right\}$$



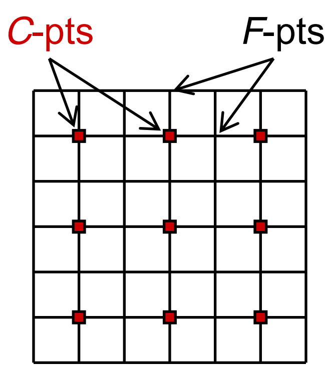
- Only makes sense if *n* is very small
  - Example: 3D Laplace on 1M procs: $\log_8(P) < 7$
- And this analysis is extremely optimistic!

Additional comments on parallel multigrid



- W-cycles scale poorly:

  $T_W \approx (2^{\log N})4\alpha + (\log N)4n\beta + (2)5n^2\gamma$

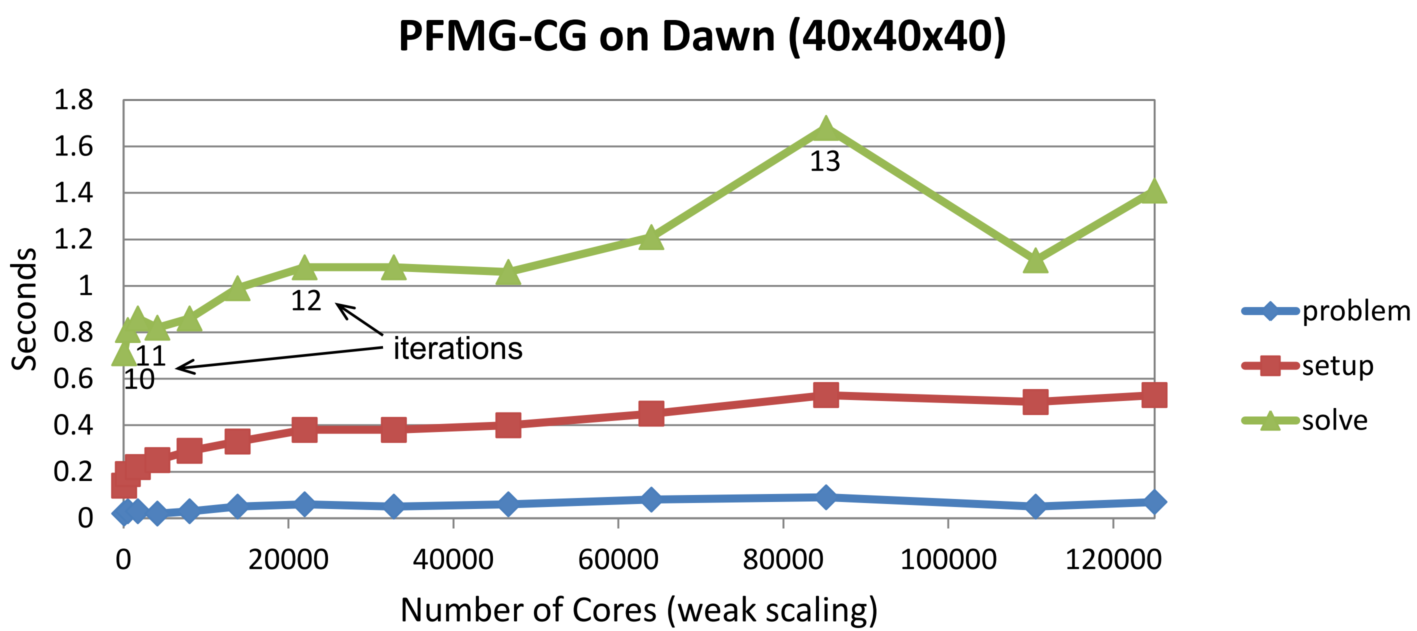


- Lexicographical Gauss-Seidel is too sequential
  - Use red/black or multi-color GS
  - Use weighted Jacobi, hybrid Jacobi/GS, L1
  - Use *C*-*F* relaxation (Jacobi on *C*-pts then *F*-pts)
  - Use Polynomial smoothers
- Parallel smoothers are often less effective

- Even if computations were free, doing extra local work can actually degrade convergence (e.g., block smoothers)

- Recent parallel multigrid papers:
  - A Survey of Parallelization Techniques for Multigrid Solvers, Chow, Falgout, Hu, Tuminaro, and Yang, *Parallel Processing For Scientific Computing*, Heroux, Raghavan, and Simon, editors, SIAM (2006)
  - Multigrid Smoothers for Ultra-Parallel Computing, Baker, Falgout, Kolev, and Yang, *SIAM J. Sci. Comput.* (to appear)

Example weak scaling results on Dawn (an IBM BG/P system at LLNL) in 2011

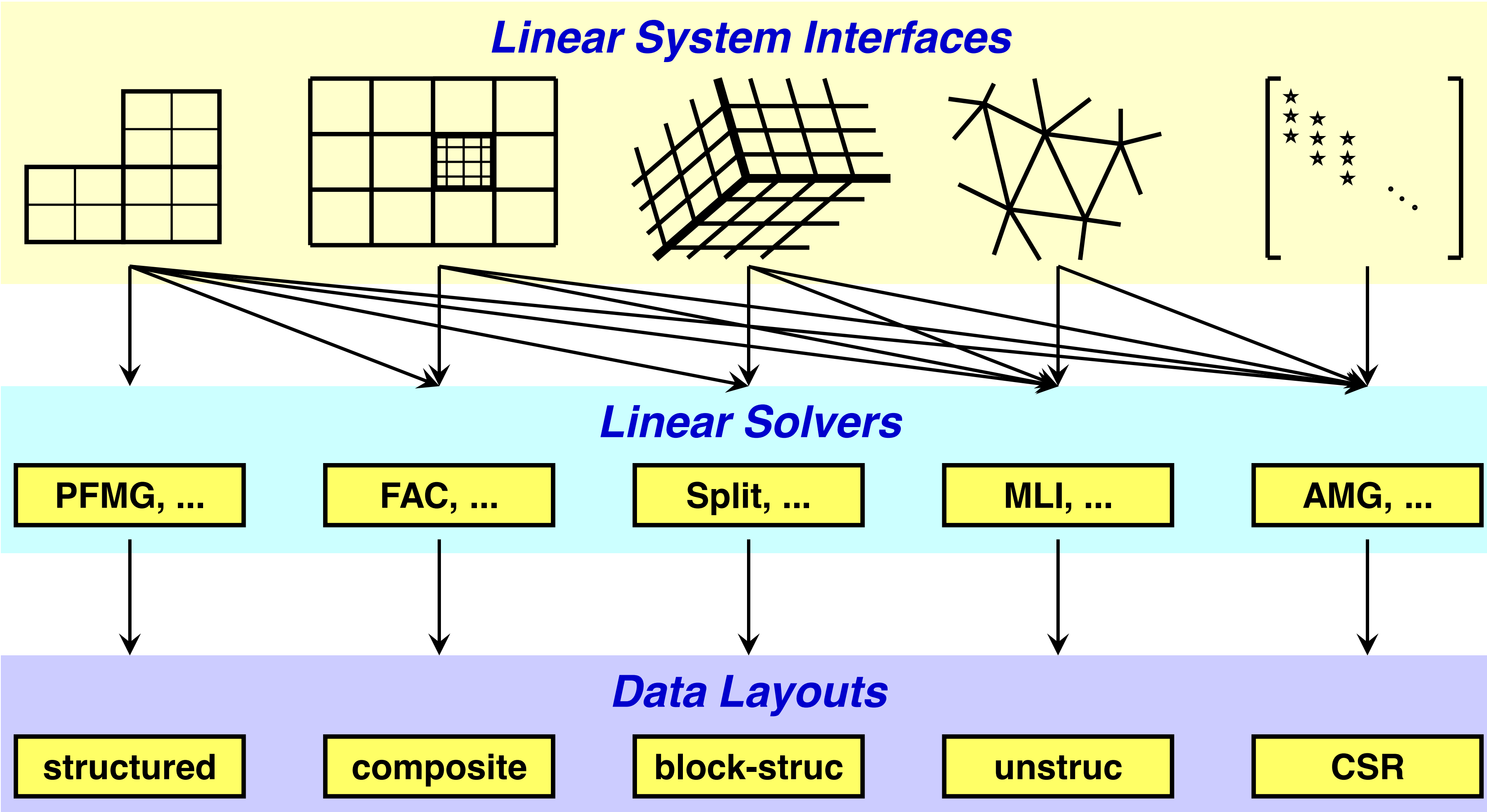


- Laplacian on a cube; 40^3 = 64K grid per processor; largest had 8 billion unknowns
- PFMG is a semicoarsening multigrid solver in *hypre*
- Constant-coefficient version - 1 trillion unknowns on 131K cores in 83 seconds
- Still room to improve setup implementation (these results already employ the assumed partition algorithm)

Multigrid Software

2007

(Conceptual) linear system interfaces are necessary to provide “best” solvers and data layouts

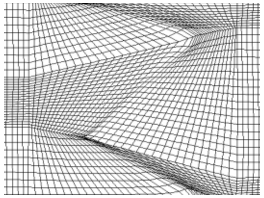


Why multiple interfaces? The key points

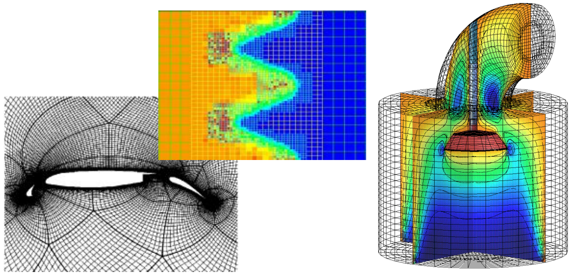
- Provides natural “views” of the linear system
- Eases some of the coding burden for users by eliminating the need to map to rows/columns
- Provides for more efficient (scalable) linear solvers
- Provides for more effective data storage schemes and more efficient computational kernels

Currently, *hypre* supports four system interfaces

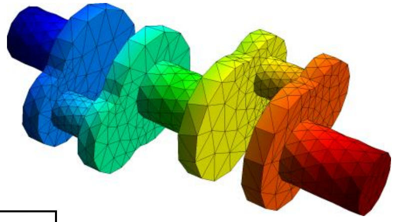
- Structured-Grid (`Struct`)
  - *logically rectangular grids*



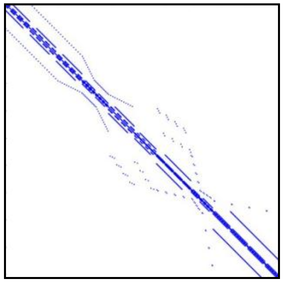
- Semi-Structured-Grid (`SStruct`)
  - *grids that are mostly structured*



- Finite Element (`FEI`)
  - *unstructured grids with finite elements*



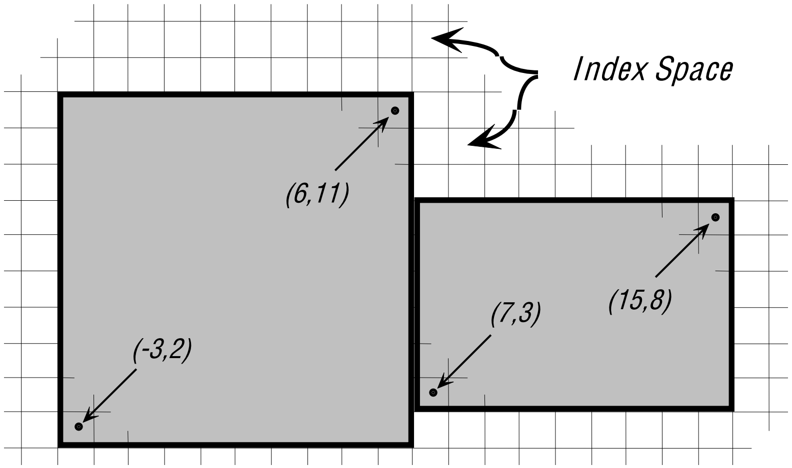
- Linear-Algebraic (`IJ`)
  - *general sparse linear systems*



- More about each next…

Structured-Grid System Interface (`Struct`)

- 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 scalar grid data is always associated with cell centers (unlike the more general SStruct interface)



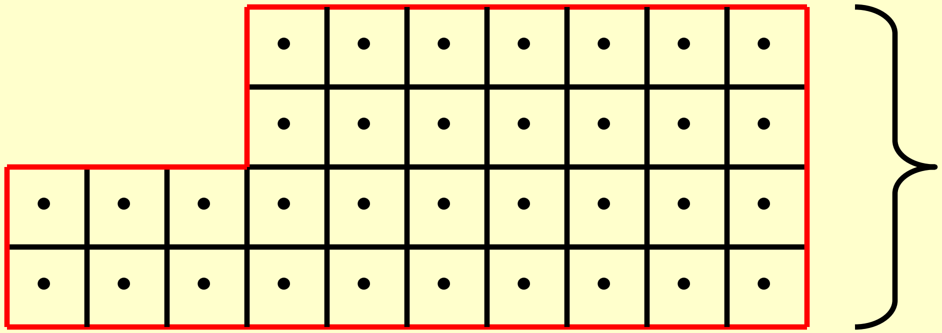
Structured-Grid System Interface (`Struct`)

- There are four basic steps involved:
  - set up the `Grid`
  - set up the `Stencil`
  - set up the `Matrix`
  - set up the right-hand-side `Vector`

- Consider the following 2D Laplacian problem

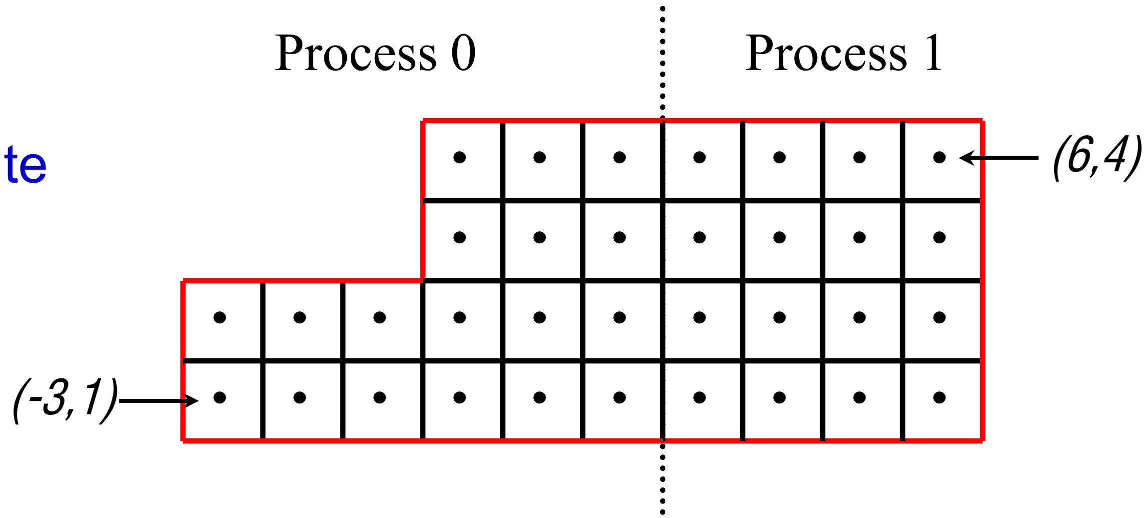
$$\begin{cases} -\nabla^2 u = f & \text{in the domain} \\ u = g & \text{on the boundary} \end{cases}$$

Structured-grid finite volume example:

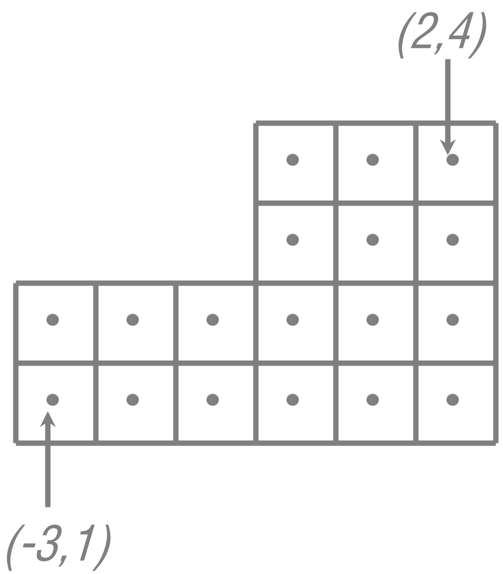


Standard 5-point finite volume discretization

Partition and distribute



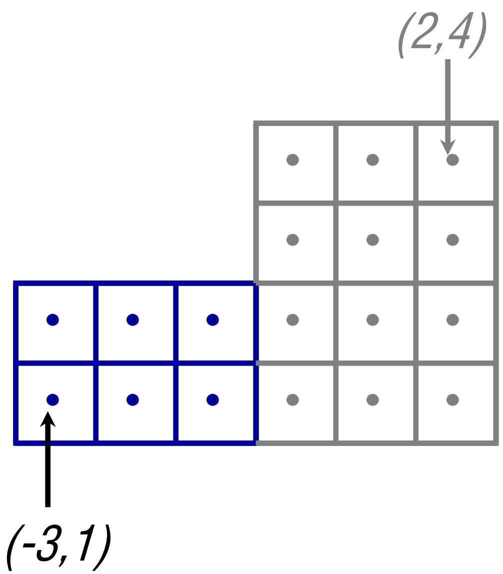
Structured-grid finite volume example:
Setting up the grid on process 0



Create the grid object

```
HYPRE_StructGrid grid;
int ndim    = 2;

HYPRE_StructGridCreate(MPI_COMM_WORLD, ndim, &grid);
```

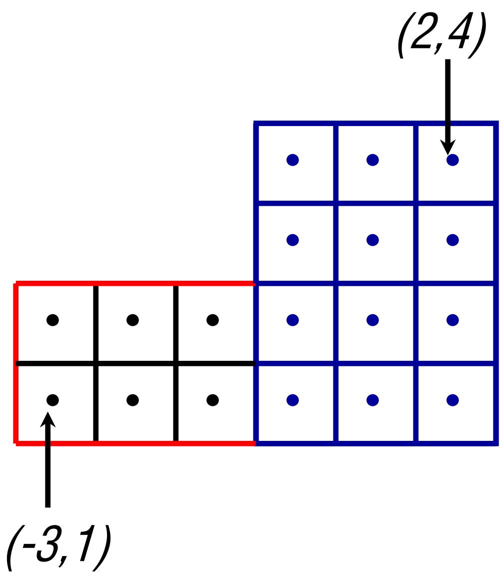
Structured-grid finite volume example:
Setting up the grid on process 0



Set grid extents for first box

```
int ilo0[2] = {-3,1};
int iup0[2] = {-1,2};

HYPRE_StructGridSetExtents(grid, ilo0, iup0);
```

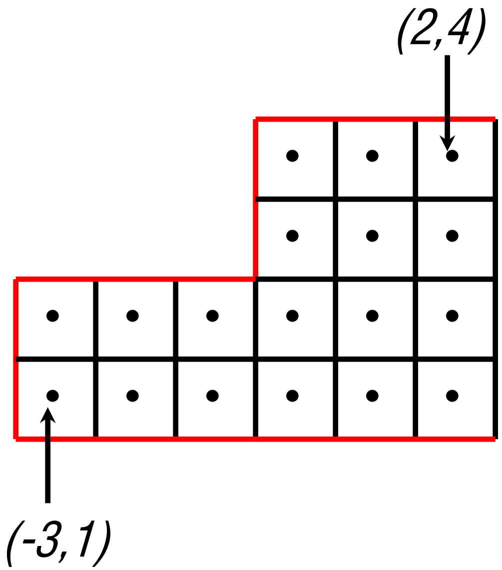
Structured-grid finite volume example:
Setting up the grid on process 0



Set grid extents for second box

```
int ilo1[2] = {0,1};
int iup1[2] = {2,4};

HYPRE_StructGridSetExtents(grid, ilo1, iup1);
```

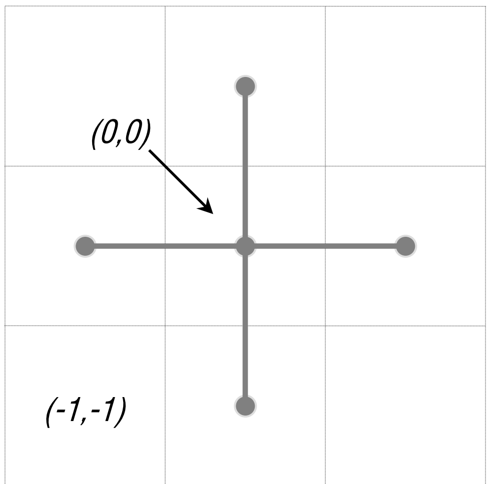
Structured-grid finite volume example:
Setting up the grid on process 0



Assemble the grid

```
HYPRE_StructGridAssemble(grid);
```

Structured-grid finite volume example:
Setting up the stencil (all processes)

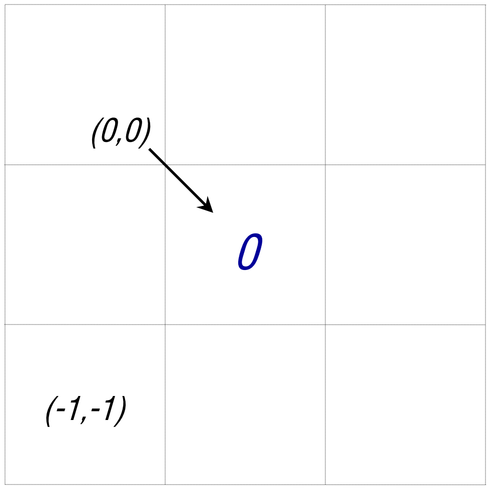


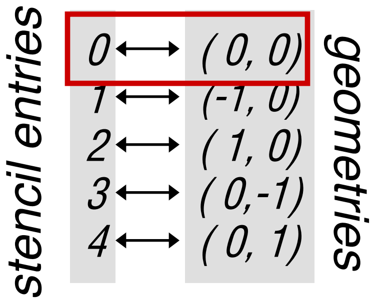
stencil entries		geometries
0	⟷	(0, 0)
1	⟷	(-1, 0)
2	⟷	(1, 0)
3	⟷	(0,-1)
4	⟷	(0, 1)

Create the stencil object

```
HYPRE_StructStencil stencil;
int ndim = 2;
int size = 5;

HYPRE_StructStencilCreate(ndim, size, &stencil);
```

Structured-grid finite volume example:
Setting up the stencil (all processes)

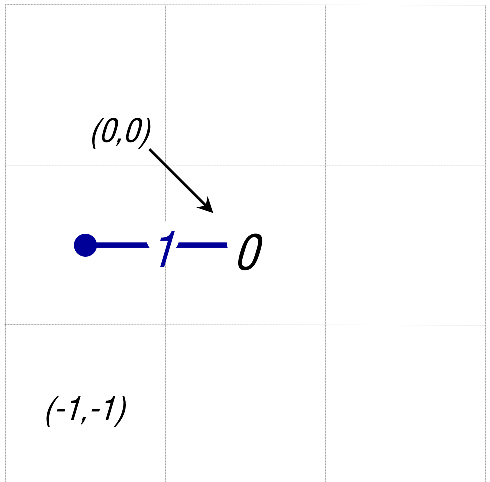


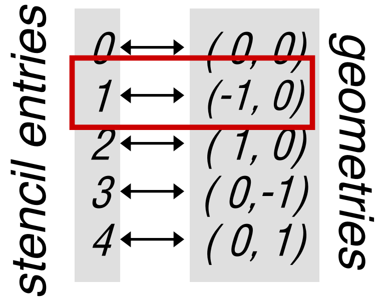


Set stencil entries

```
int entry = 0;
int offset[2] = {0,0};

HYPRE_StructStencilSetElement(stencil, entry, offset);
```

Structured-grid finite volume example:
Setting up the stencil (all processes)

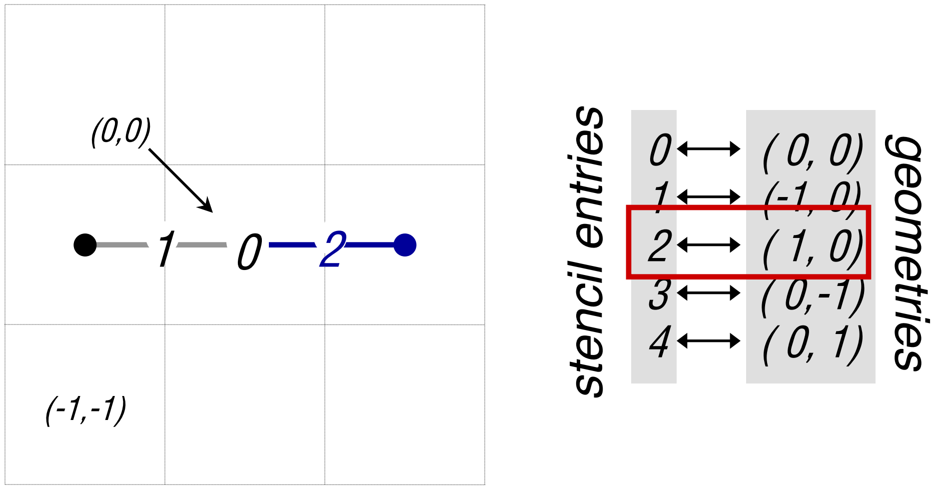




Set stencil entries

```
int entry = 1;
int offset[2] = {-1,0};

HYPRE_StructStencilSetElement(stencil, entry, offset);
```

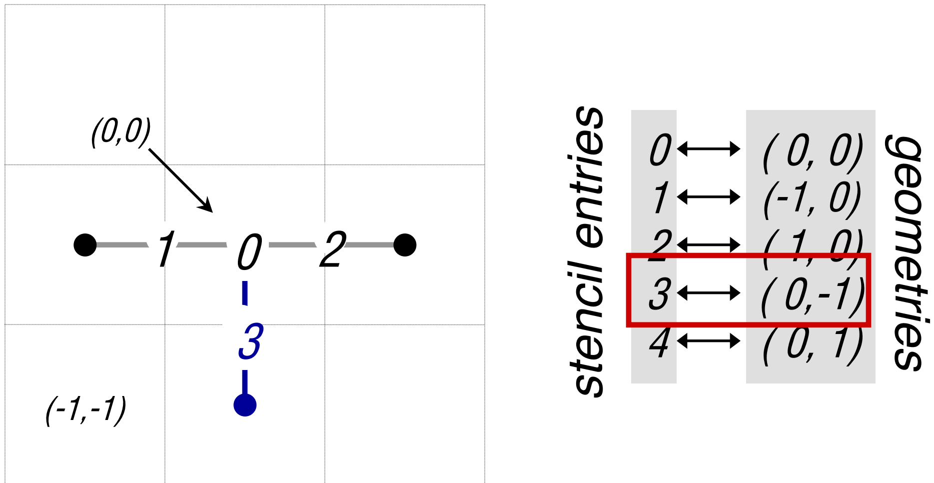
Structured-grid finite volume example:
Setting up the stencil (all processes)



Set stencil entries

```
int entry = 2;
int offset[2] = {1,0};

HYPRE_StructStencilSetElement(stencil, entry, offset);
```

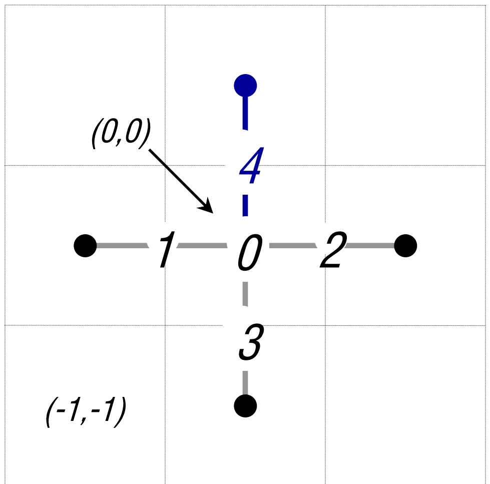
Structured-grid finite volume example:
Setting up the stencil (all processes)

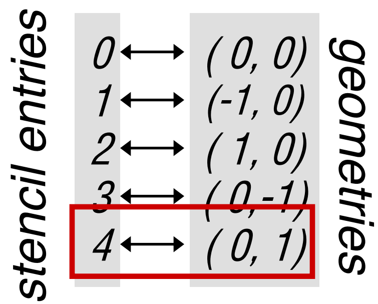


Set stencil entries

```
int entry = 3;
int offset[2] = {0,-1};

HYPRE_StructStencilSetElement(stencil, entry, offset);
```

Structured-grid finite volume example:
Setting up the stencil (all processes)

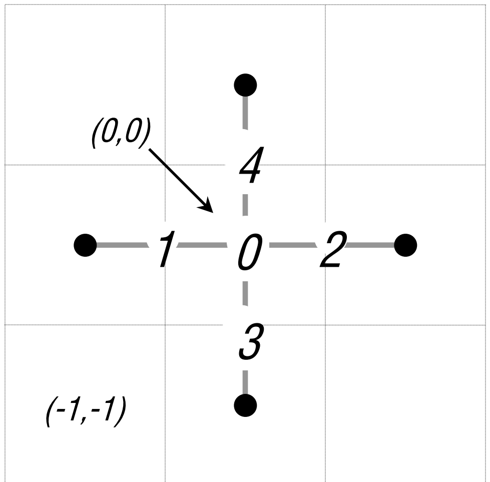


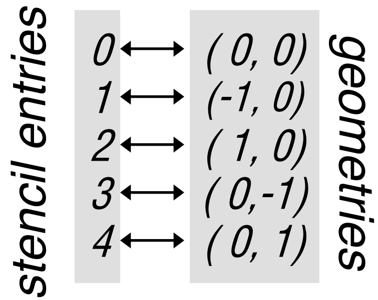


Set stencil entries

```
int entry = 4;
int offset[2] = {0,1};

HYPRE_StructStencilSetElement(stencil, entry, offset);
```

Structured-grid finite volume example:
Setting up the stencil (all processes)

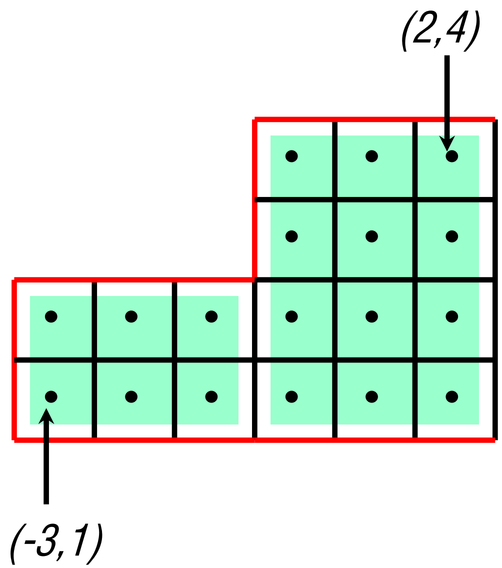




That’s it!

There is no assemble routine

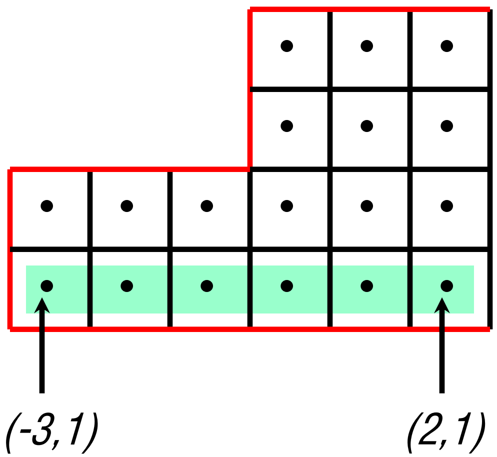
Structured-grid finite volume example :
Setting up the matrix on process 0



$$\begin{bmatrix} & S4 & \\ S1 & S0 & S2 \\ & S3 & \end{bmatrix} = \begin{bmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{bmatrix}$$

```
HYPRE_StructMatrix  A;
double  vals[24] = {4, -1, 4, -1, …};
int  nentries = 2;
int  entries[2] = {0,3};

HYPRE_StructMatrixCreate(MPI_COMM_WORLD,
   grid, stencil, &A);
HYPRE_StructMatrixInitialize(A);

HYPRE_StructMatrixSetBoxValues(A,
   ilo0, iup0, nentries, entries, vals);
HYPRE_StructMatrixSetBoxValues(A,
   ilo1, iup1, nentries, entries, vals);

/* set boundary conditions */
…
HYPRE_StructMatrixAssemble(A);
```

Structured-grid finite volume example :
Setting up the matrix bc's on process 0

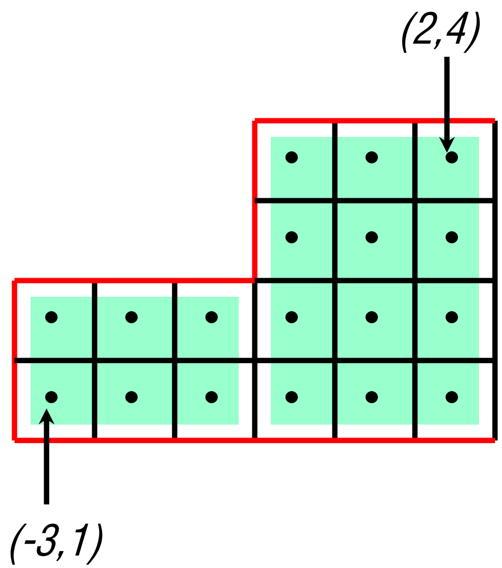


$$\begin{bmatrix} & S4 & \\ S1 & S0 & S2 \\ & \color{red}{S3} & \end{bmatrix} = \begin{bmatrix} & -1 & \\ -1 & 4 & -1 \\ & \color{red}{0} & \end{bmatrix}$$

```
int  ilo[2] = {-3, 1};
int  iup[2] = { 2, 1};
double  vals[6] = {0, 0, …};
int nentries = 1;

/* set interior coefficients */
…

/* implement boundary conditions */
…

i = 3;
HYPRE_StructMatrixSetBoxValues(A,
   ilo, iup, nentries, &i, vals);

/* complete implementation of bc's */
…
```

A structured-grid finite volume example :

Setting up the right-hand-side vector on process 0



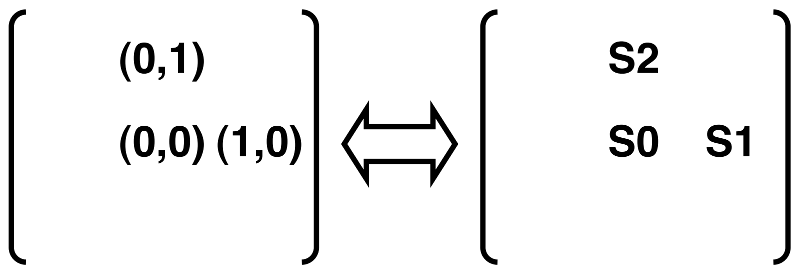

```
HYPRE_StructVector  b;
double  vals[12] = {0, 0, …};

HYPRE_StructVectorCreate(MPI_COMM_WORLD,
   grid, &b);
HYPRE_StructVectorInitialize(b);

HYPRE_StructVectorSetBoxValues(b,
   ilo0, iup0, vals);
HYPRE_StructVectorSetBoxValues(b,
   ilo1, iup1, vals);

HYPRE_StructVectorAssemble(b);
```

Symmetric Matrices

- Some solvers support symmetric storage
- Between `Create()` and `Initialize()`, call:

```
HYPRE_StructMatrixSetSymmetric(A, 1);
```

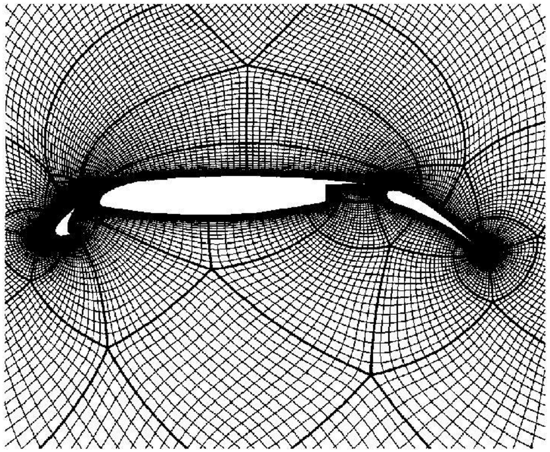
- For best efficiency, only set half of the coefficients



- This is enough info to recover the full 5-pt stencil

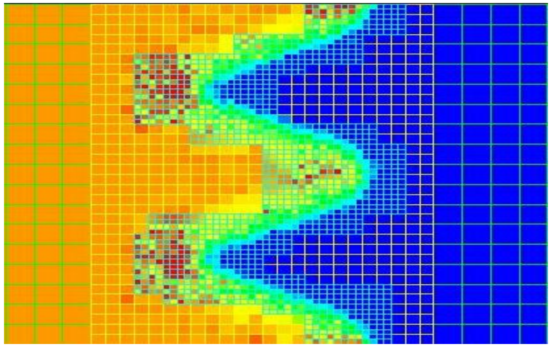
Semi-Structured-Grid System Interface (`SStruct`)

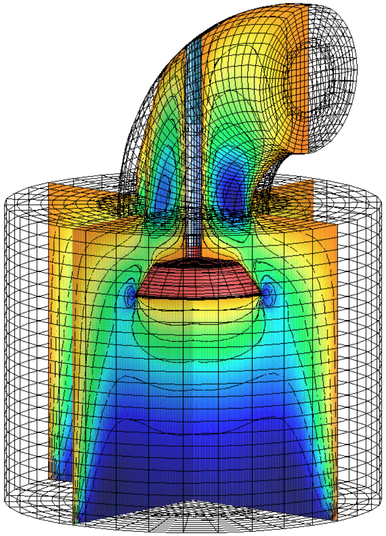
- Allows more general grids:
  - Grids that are mostly (but not entirely) structured
  - Examples: *block-structured grids, structured adaptive mesh refinement grids, overset grids*



Block-Structured

Adaptive Mesh Refinement

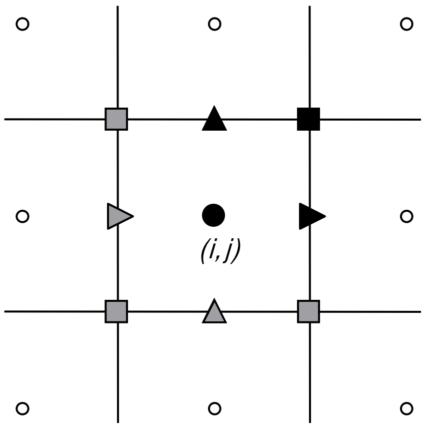




Overset

Semi-Structured-Grid System Interface (`SStruct`)

- Allows more general PDE's
  - Multiple variables (system PDE's)
  - Multiple variable types (cell centered, face centered, vertex centered, …)



Variables are referenced by the abstract cell-centered index to the left and down

Building different matrix/vector storage formats with the `SStruct` interface

- Efficient preconditioners often require specific matrix/vector storage schemes

- Between `Create()` and `Initialize`(), call:

```
HYPRE_SStructMatrixSetObjectType(A, HYPRE_PARCSR);
```

- After `Assemble()`, call:

```
HYPRE_SStructMatrixGetObject(A, &parcsr_A);
```

- Now, use the `ParCSR` matrix with compatible solvers such as BoomerAMG (algebraic multigrid)

Current solver / preconditioner availability via *hypre*'s linear system interfaces

Data Layouts	Solvers	System Interfaces: Struct	SStruct	FEI	IJ
Structured	Jacobi	✓	✓		
	SMG	✓	✓		
	PFMG	✓	✓		
Semi-structured	Split		✓		
	SysPFMG		✓		
	FAC		✓		
	Maxwell		✓		
Sparse matrix	AMS		✓	✓	✓
	BoomerAMG		✓	✓	✓
	MLI		✓	✓	✓
	ParaSails		✓	✓	✓
	Euclid		✓	✓	✓
	PILUT		✓	✓	✓
Matrix free	PCG	✓	✓	✓	✓
	GMRES	✓	✓	✓	✓
	BiCGSTAB	✓	✓	✓	✓
	Hybrid	✓	✓	✓	✓

Setup and use of solvers is largely the same (*see Reference Manual for details*)

- Create the solver

```
HYPRE_SolverCreate(MPI_COMM_WORLD, &solver);
```

- Set parameters

```
HYPRE_SolverSetTol(solver, 1.0e-06);
```

- Prepare to solve the system

```
HYPRE_SolverSetup(solver, A, b, x);
```

- Solve the system

```
HYPRE_SolverSolve(solver, A, b, x);
```

- Get solution info out via system interface

```
HYPRE_StructVectorGetValues(struct_x, index,
values);
```

- Destroy the solver

```
HYPRE_SolverDestroy(solver);
```

Solver example: SMG-PCG

```
/* define preconditioner (one symmetric V(1,1)-cycle) */
HYPRE_StructSMGCreate(MPI_COMM_WORLD, &precond);
HYPRE_StructSMGSetMaxIter(precond, 1);
HYPRE_StructSMGSetTol(precond, 0.0);
HYPRE_StructSMGSetZeroGuess(precond);
HYPRE_StructSMGSetNumPreRelax(precond, 1);
HYPRE_StructSMGSetNumPostRelax(precond, 1);

HYPRE_StructPCGCreate(MPI_COMM_WORLD, &solver);
HYPRE_StructPCGSetTol(solver, 1.0e-06);

/* set preconditioner */
HYPRE_StructPCGSetPrecond(solver,
   HYPRE_StructSMGSolve, HYPRE_StructSMGSetup, precond);

HYPRE_StructPCGSetup(solver, A, b, x);
HYPRE_StructPCGSolve(solver, A, b, x);
```

USQCD / FASTMath Interactions

- Changes to hypre (none of these are difficult to do, but were just never a priority before)
  - Extend SStruct beyond 3D
  - Add support for complex
  - Implement an adaptive AMG method
  - Provide support for building PETSc and Trilinos matrices
- PETSc and Trilinos differ from hypre in that they are more like math toolboxes for parallel computing
  - Both have linear solvers and multigrid methods
  - Both provide access to hypre's BoomerAMG solver
  - Both have nonlinear solvers and time integration methods
- SUNDIALS is another FASTMath package (with a long long history at LLNL) that has nonlinear solvers and time integration methods (in addition to ODE solvers and sensitivity analysis methods)

USQCD / FASTMath Interactions – benefits of coupling to external libraries

- NOT for achieving the absolute best performance of a particular algorithm
  - An algorithm can always be implemented more efficiently directly in the code that uses it
  - An algorithm can always be tweaked, tuned, and tailored more effectively directly in the application code
  - This is because applications are not hampered by library interfaces and the need to support other applications

- Given this…

- Excellent performance is highly achievable with libraries
- You benefit directly from other people's research (possibly done in the context of a completely different application)
- New algorithms research is often easier to do
- You have more time to focus on science

Thank You!

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Adaptive *AMG* idea: use the method to improve the method

- Requires no a-priori knowledge of the near null space
- Idea: uncover *representatives* of slowly-converging error by applying the "current method" to $Ax = 0$, then use these to adapt (improve) the method

- Achi Brandt's *Bootstrap AMG* is an adaptive method
- PCG can be viewed as an adaptive method
  - Not optimal because it uses a global view
  - The key is to view representatives locally

- We developed 2 methods: αAMG and αSA (SISC pubs)

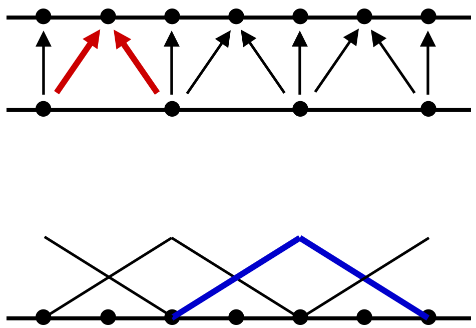
To build effective interpolation, it is important to interpret the near null space in a local way

- (2-level) Coarse-grid correction is a projection

$$(I - P(P^T A P)^{-1} P^T A)e$$

- Better to break up near null space into a local basis

$$P = \frac{1}{2}\begin{bmatrix} 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \\ 2 \end{bmatrix} \quad \Rightarrow \quad P = \frac{1}{2}\begin{bmatrix} 2 & & & \\ 1 & 1 & & \\ & 2 & & \\ & 1 & 1 & \\ & & 2 & \\ & & 1 & 1 \\ & & & 2 \end{bmatrix}$$



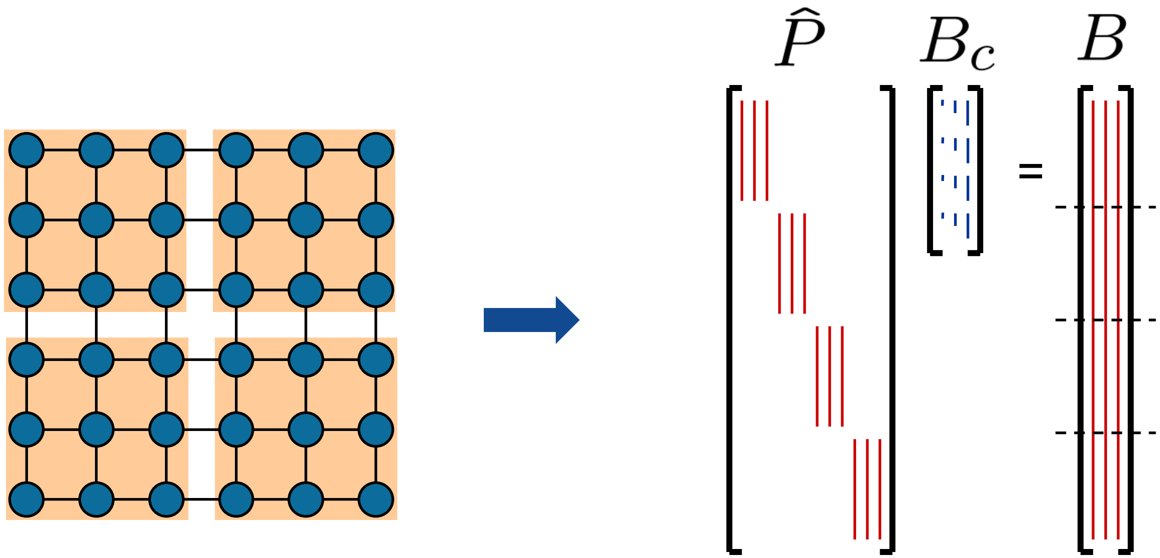
Deflation – not optimal

Multigrid – optimal

- Get full approximation property (low-frequency Fourier modes in this example)

SA builds interpolation by first chopping up a global basis, then smoothing it

- Tentative interpolation is constructed from “aggregates” (local QR factorization is used to orthonormalize)



$$\hat{P}^T \hat{P} = I$$

- Smoothing adds basis overlap and improves approximation property

$$P = S\hat{P}$$

Adaptive smoothed aggregation (αSA) automatically builds the global basis for SA

- Generate the basis one vector at a time
  - Start with relaxation on $Au=0$ → u_1 → αSA(u_1)
  - Use αSA(u_1) on $Au=0$ → u_2 → αSA(u_1,u_2)
  - Etc., until we have a good method

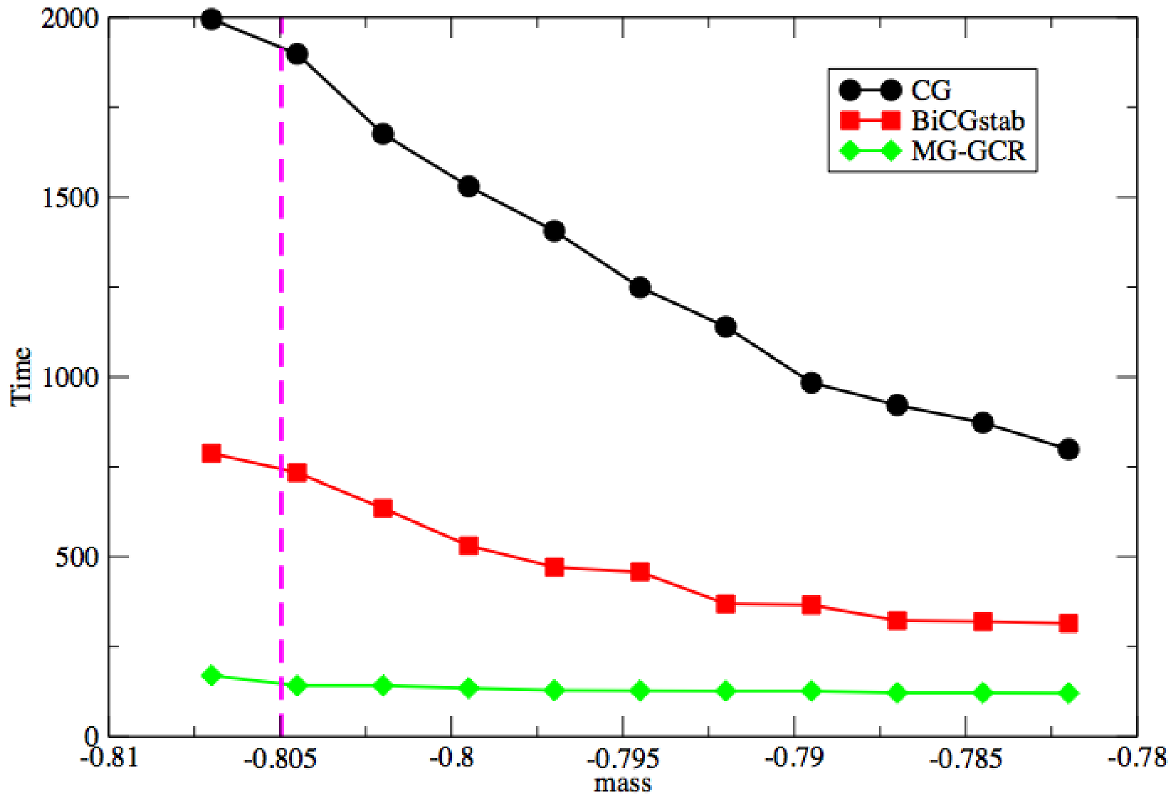
- Setup is expensive, but is amortized over many RHS's

- Published in 2004, highlighted in SIAM Review in 2005
  - Brezina, Falgout, MacLachlan, Manteuffel, McCormick, and Ruge, "Adaptive smoothed aggregation (αSA)," *SIAM J. Sci. Comput.* (2004)
- Successfully applied to 2D QED
  - Brannick, Brezina, Keyes, Livne, Livshits, MacLachlan, Manteuffel, McCormick, Ruge, and Zikatanov, "Adaptive smoothed aggregation in lattice QCD," Springer (2006)

Customized αSA takes advantage of the regular geometry and unitary connections in QCD

- Uses regular geometric blocking (e.g., $4^d \times \frac{1}{2}N_s \times N_c$)
- Developed two methods:
  - $D^\dagger D$-MG solves the normal equations (HPD)
  - D-MG solves original system (not HPD)
- $D^\dagger D$-MG requires more vectors than D-MG, so it is more expensive
- D-MG can use $R = P^T$, even though D is not Hermitian
  - Coarse operator looks like Dirac and retains γ_5 Hermiticity
- D-MG uses Minimum Residual for relaxation

4D Wilson-Dirac Results: D-MG shows no critical slowing down (Time)



- Parameters: $N=16^3x32$, $\beta=6.0$, $m_{crit} = -0.8049$
- D-MG Parameters: 4^4x3x2 blocking, 3 levels, W(2,2,4) cycle, $N_v = 20$, setup run at m_{crit}