



**Robert D. Falgout**  
*Center for Applied Scientific Computing*

**September 5, 2002**



# *hypre* team

---

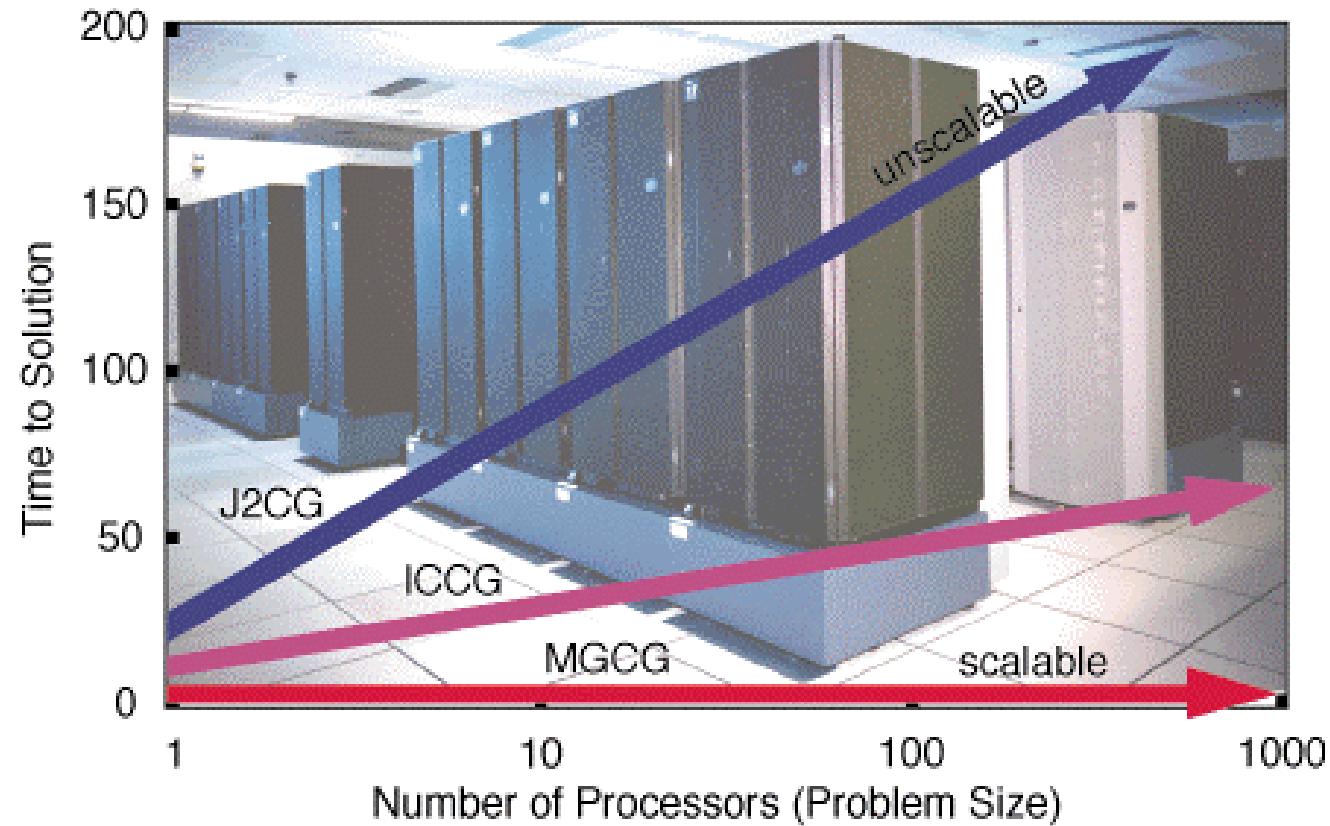
- Rob Falgout (project leader)
- Guillermo Castilla
- Edmond Chow
- Andy Cleary
- Van Emden Henson
- Jim Jones
- Mike Lambert
- Barry Lee
- Jeff Painter
- Charles Tong
- Tom Treadway
- Panayot Vassilevski
- Ulrike Meier Yang

# Outline

---

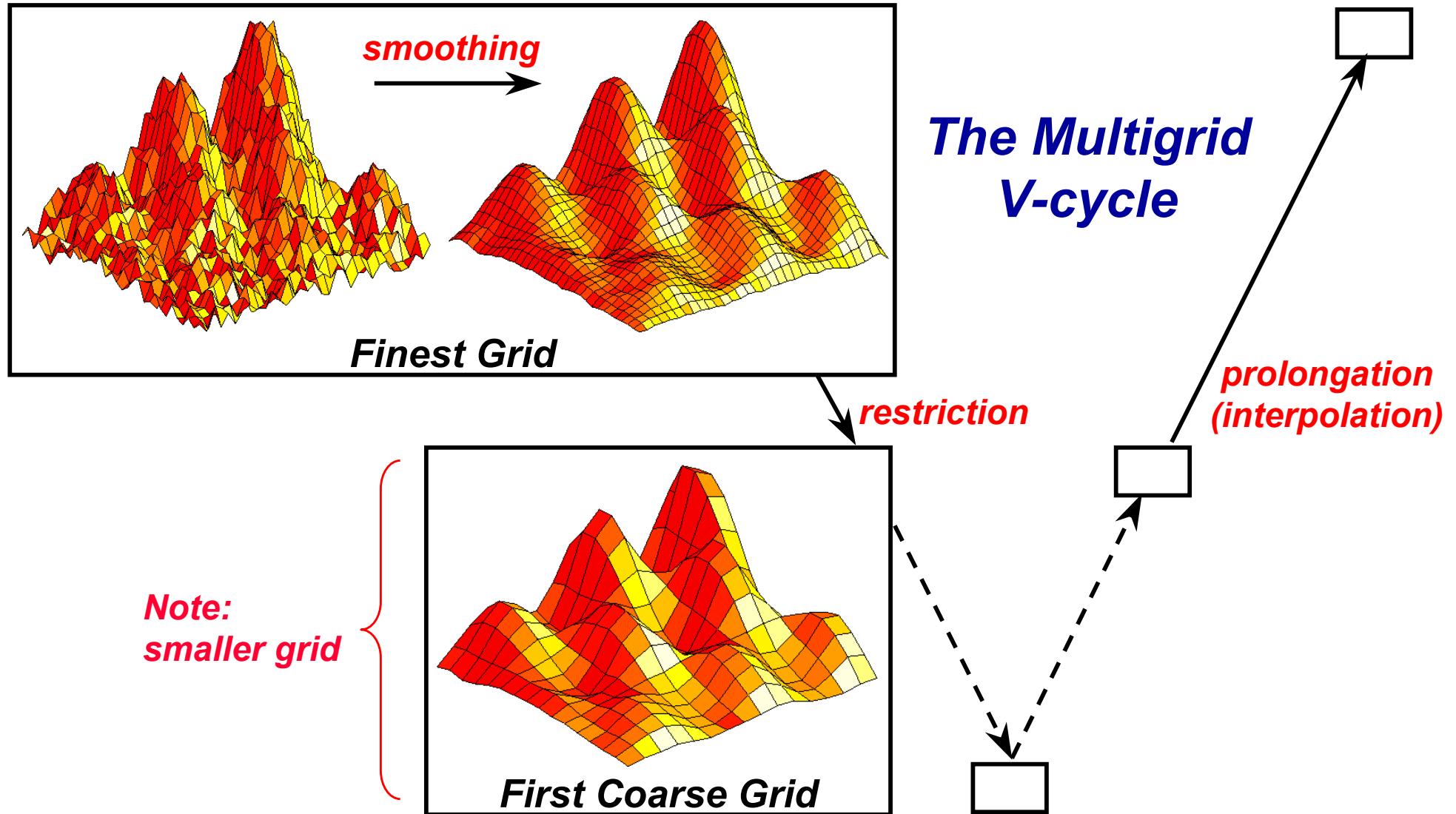
- **Introduction / Motivation**
- **Getting Started / Conceptual Interfaces**
  
- **Structured-Grid Interface (Struct)**
- **Semi-Structured-Grid Interface (SStruct)**
- **Finite Element Interface (FEI)**
- **Linear-Algebraic Interface (IJ)**
  
- **Solvers and Preconditioners**
- **Additional Information**

# Scalability is a central issue for large-scale parallel computing



Linear solver convergence can be discussed independent of parallel computing, and is often overlooked as a key scalability issue.

# Multigrid uses coarse grids to efficiently damp out smooth error components

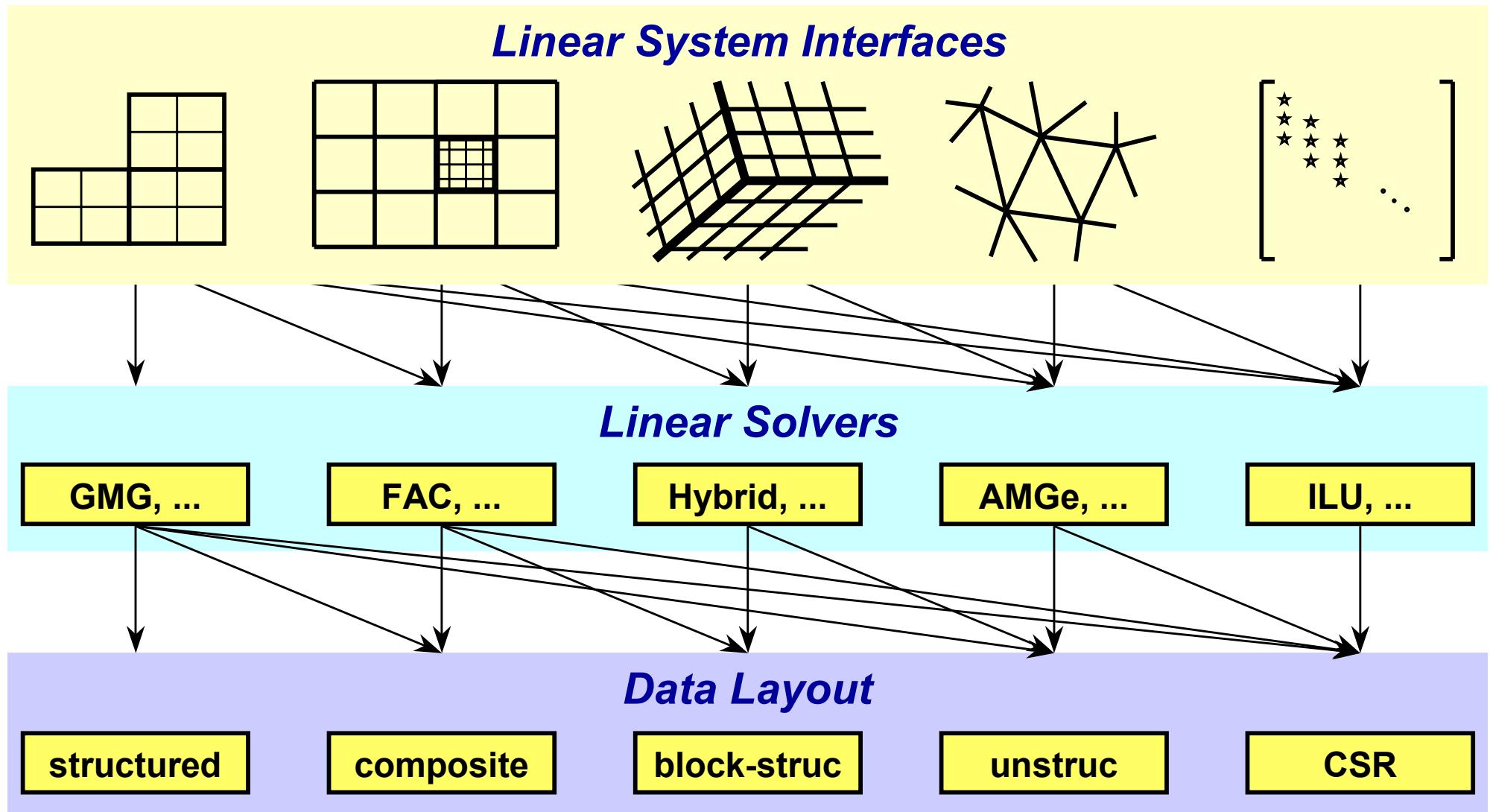


# Getting Started

---

- **Before writing your code:**
  - choose a conceptual interface
  - choose a solver / preconditioner
  - choose a matrix type that is compatible with your solver / preconditioner and conceptual interface
  
- **Now write your code:**
  - build auxiliary structures (e.g., grids, stencils)
  - build matrix/vector through conceptual interface
  - build solver/preconditioner
  - solve the system
  - get desired information from the solver

# Multiple 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

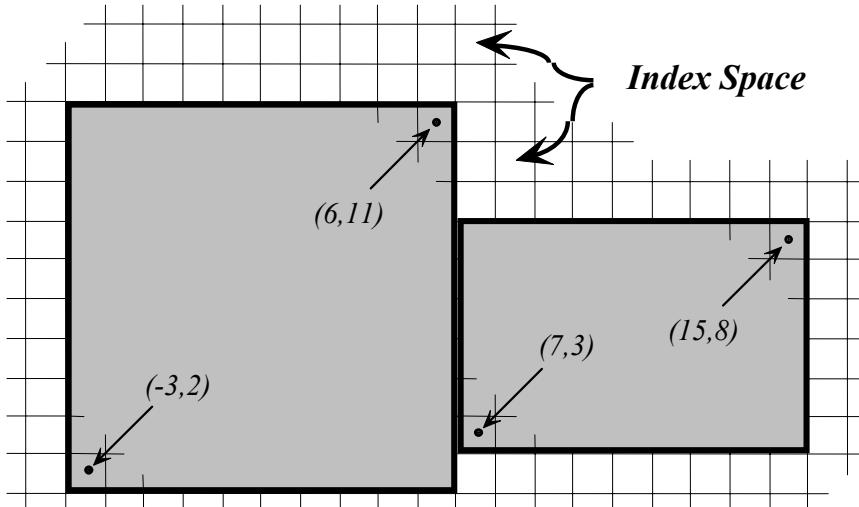
# *hypre currently supports four conceptual interfaces*

---

- **Structured-Grid Interface (`Struct`)**
  - *applications with logically rectangular grids*
- **Semi-Structured-Grid Interface (`SStruct`)**
  - *applications with grids that are mostly—but not entirely—structured (e.g., block-structured, structured AMR, overset)*
- **Finite Element Interface (`FEI`)**
  - *unstructured-grid, finite element applications*
- **Linear-Algebraic Interface (`IJ`)**
  - *applications with 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)

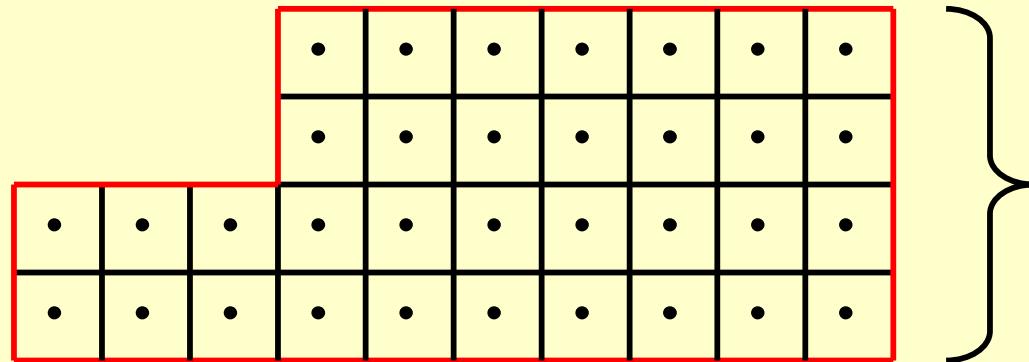
# Structured-Grid System Interface (Struct)

---

- 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

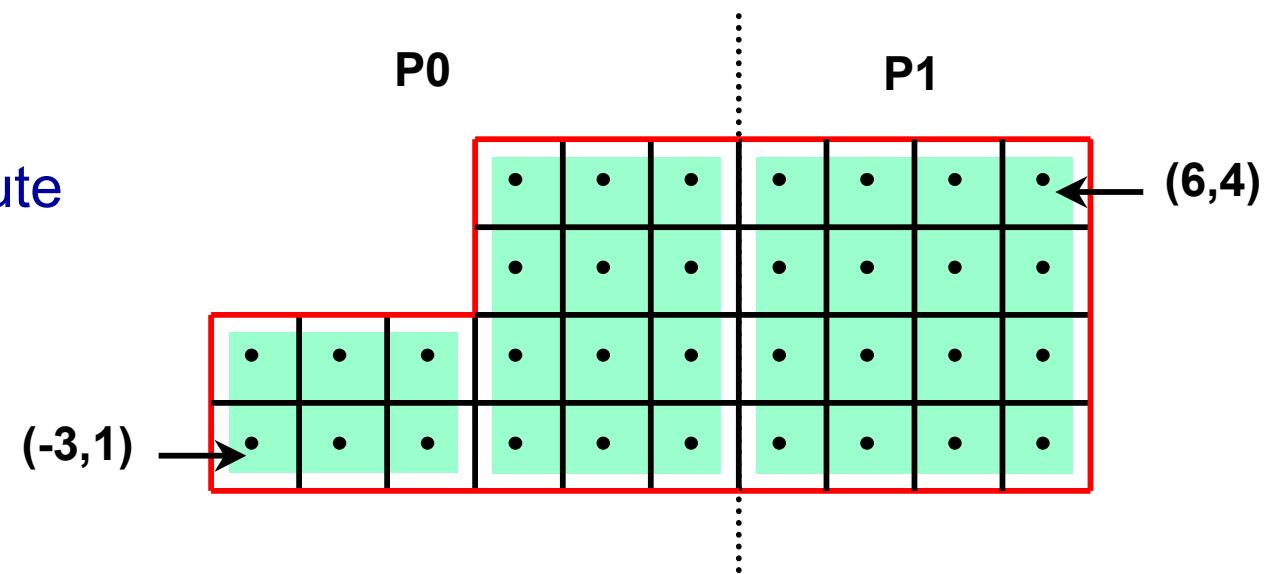
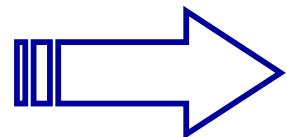
$$\begin{cases} \nabla^2 u = f, & \text{in the domain} \\ u = 0, & \text{on the boundary} \end{cases}$$

# A structured-grid finite volume example:



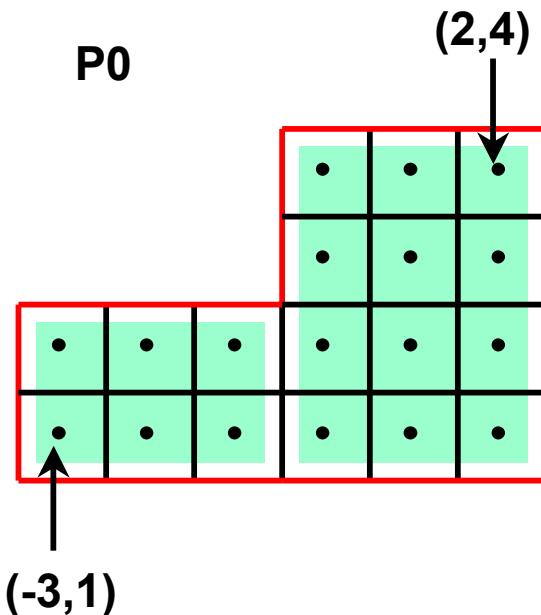
Standard 5-point finite volume discretization

Partition and distribute



# A structured-grid finite volume example : *setting up the Grid*

---



```
HYPRE_StructGrid  grid;
int  ndim = 2;
int  ilo[2][2] = {{-3, 1}, {0, 1}};
int  iup[2][2] = {{-1, 2}, {2, 4}};

HYPRE_StructGridCreate(MPI_COMM_WORLD,
                      ndim, &grid);

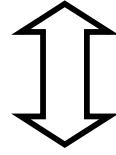
HYPRE_StructGridSetExtents(grid,
                           ilo[0], iup[0]);
HYPRE_StructGridSetExtents(grid,
                           ilo[1], iup[1]);

HYPRE_StructGridAssemble(grid);
```

# A structured-grid finite volume example : *setting up the Stencil*

---

$$\begin{bmatrix} & (0,1) \\ (-1,0) & (0,0) & (1,0) \\ & (0,-1) \end{bmatrix}$$



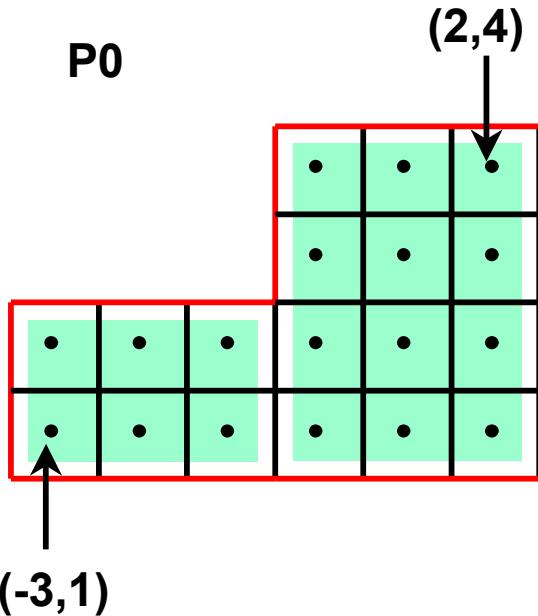
$$\begin{bmatrix} & S4 \\ S1 & S0 & S2 \\ & S3 \end{bmatrix}$$

```
HYPRE_StructStencil  stencil;
int  ndim = 2;
int  size = 5;
int  offsets[5][2] = {{0,0},
                      {-1,0}, {1,0},
                      {0,-1}, {0,1}};
int  s;

HYPRE_StructStencilCreate(ndim, size,
                         &stencil);

for (s = 0; s < 5; s++)
{
    HYPRE_StructStencilSetElement(
        stencil, s, offsets[s]);
}
```

# A structured-grid finite volume example : *setting up the Matrix*



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

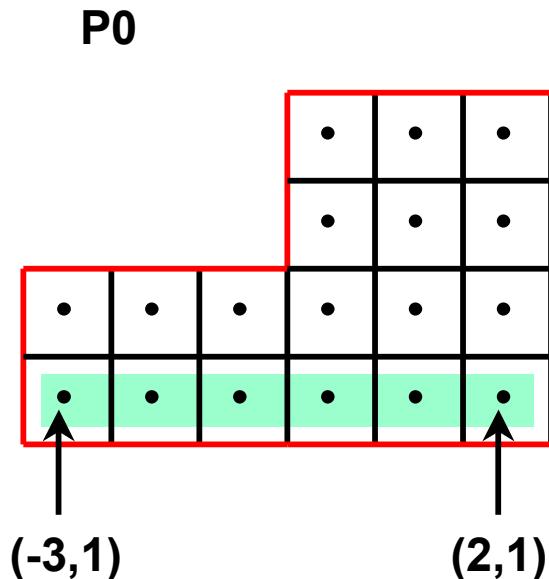
```
HYPRE_StructMatrix A;
double values[36] = {4, -1, 4, -1, ...};
int nentries = 2;
int entries[2] = {0, 3};

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

HYPRE_StructMatrixSetBoxValues(A,
    ilo[0], iup[0], nentries, entries,
    values);
HYPRE_StructMatrixSetBoxValues(A,
    ilo[1], iup[1], nentries, entries,
    values);

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

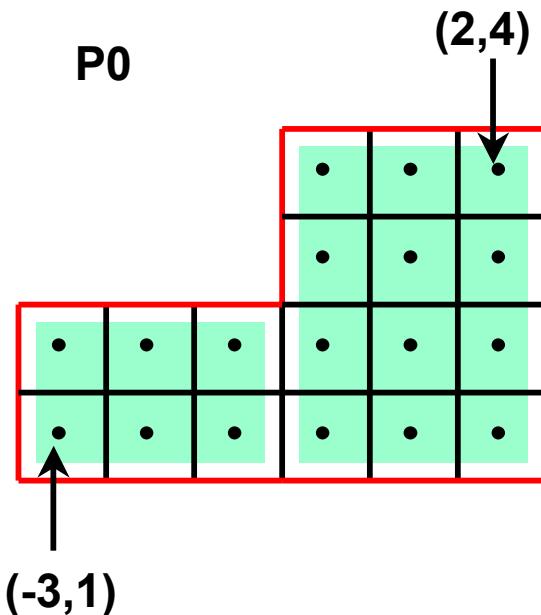
# A structured-grid finite volume example : setting up the Matrix (bc's)



```
int ilo[2] = {-3, 1};  
int iup[2] = { 2, 1};  
double values[12] = {0, 0, ...};  
int nentries = 1;  
  
/* set interior coefficients */  
...  
  
/* implement boundary conditions */  
...  
  
i = 3;  
HYPRE_StructMatrixSetBoxValues(A,  
    ilo, iup, nentries, &i, values);  
  
/* complete implementation of bc's */  
...
```

# A structured-grid finite volume example : *setting up the right-hand-side Vector*

---



```
HYPRE_StructVector b;  
double values[18] = {0, 0, ...};  
  
HYPRE_StructVectorCreate(MPI_COMM_WORLD,  
    grid, &b);  
HYPRE_StructVectorInitialize(b);  
  
HYPRE_StructVectorSetBoxValues(b,  
    ilo[0], iup[0], values);  
HYPRE_StructVectorSetBoxValues(b,  
    ilo[1], iup[1], values);  
  
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

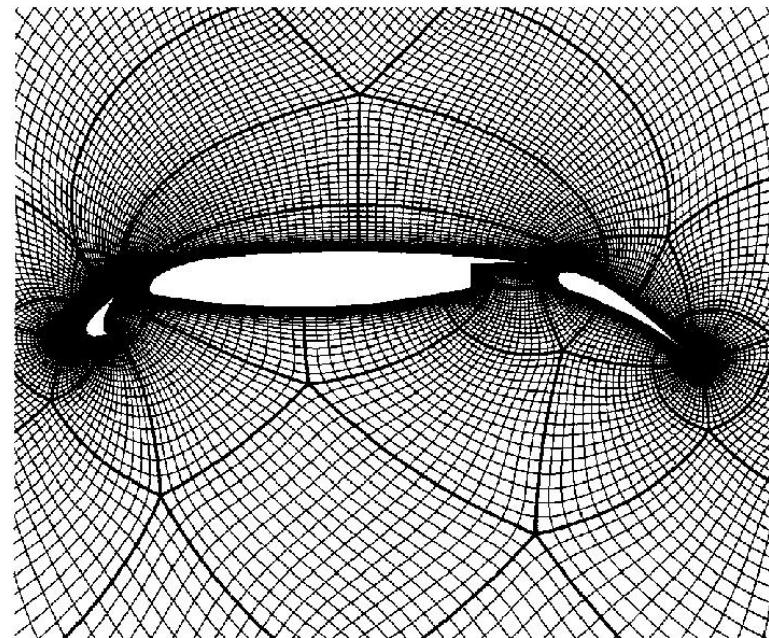
$$\begin{bmatrix} & (0,1) \\ (0,0) & (1,0) \end{bmatrix} \iff \begin{bmatrix} & s_2 \\ s_0 & s_1 \end{bmatrix}$$

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

# Semi-Structured-Grid System Interface (SStruct)

---

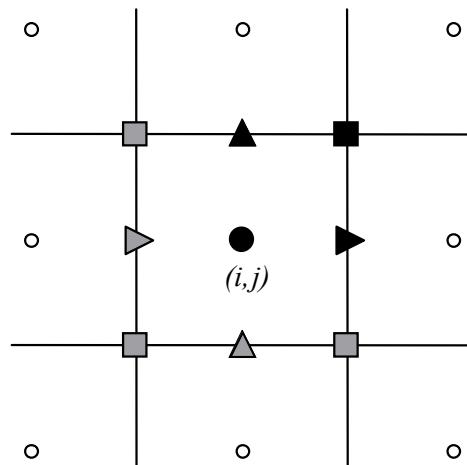
- Allows more general grids
  - Grids that are mostly—but not entirely—structured
  - Examples: **block-structured grids, structured adaptive mesh refinement grids, overset grids**



# 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

# Semi-Structured-Grid System Interface (SStruct)

---

- The SStruct grid is composed out of a number of structured grid *parts*
- The interface uses a *graph* to allow nearly arbitrary relationships between part data
- The graph is constructed from stencils plus some additional data-coupling information set either
  - directly with `GraphAddEntries()`, or
  - by relating parts with `GridSetNeighborBox()`
- We will consider two examples:
  - block-structured grid
  - structured adaptive mesh refinement

# Semi-Structured-Grid System Interface (SStruct)

---

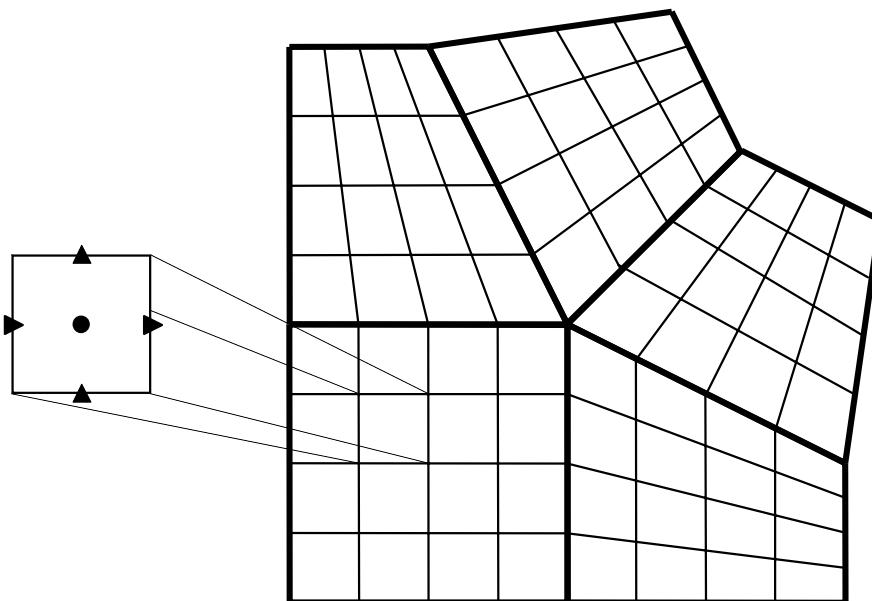
- There are five basic steps involved:
  - set up the Grid
  - set up the Stencils
  - **set up the Graph**
  - set up the Matrix
  - set up the right-hand-side vector

# Block-structured grid example (SStruct)

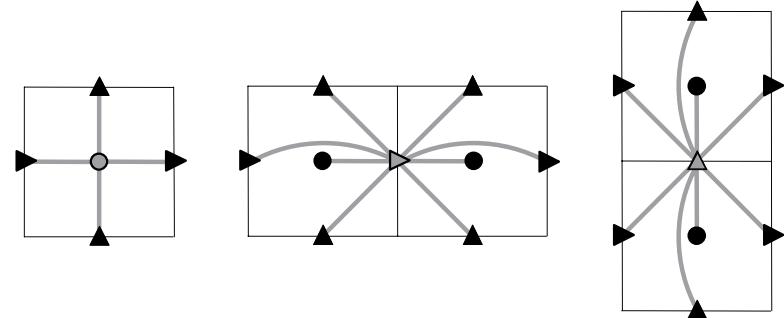
- Consider the following block-structured grid discretization of the diffusion equation

$$-\nabla \cdot (D \nabla u) + \sigma u = f$$

A block-structured grid with  
3 variable types

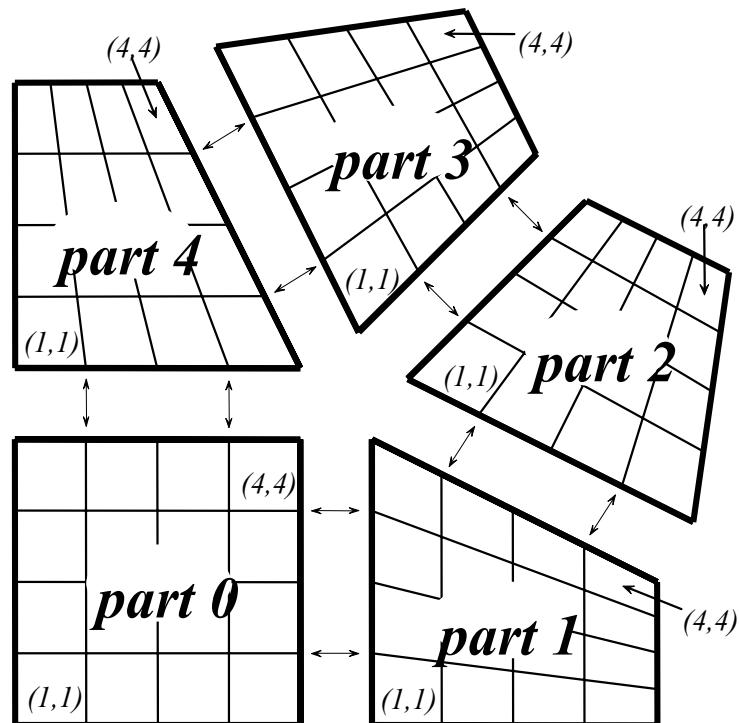


The 3 discretization stencils



# Block-structured grid example (SStruct)

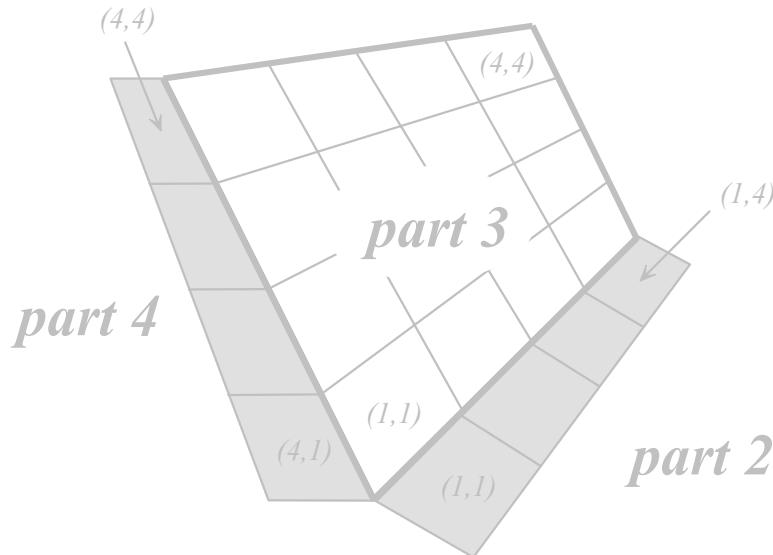
- The Grid is described in terms of 5 separate logically-rectangular parts



- Here, we assume 5 processes such that process  $p$  owns part  $p$  (note: user determines the distribution)
- We consider the interface calls made by process 3

# Block-structured grid example:

## Setting up the grid on process 3



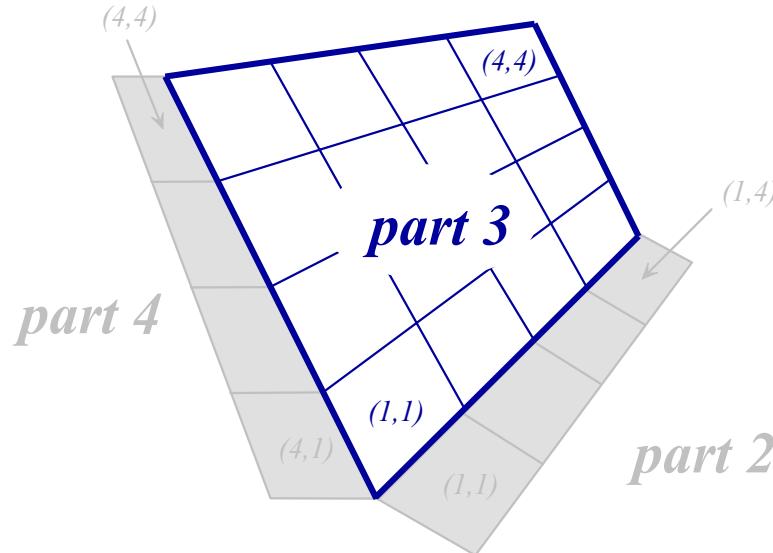
Create the grid object

```
HYPRE_SStructGrid grid;  
int ndim    = 2;  
int nparts  = 5;
```

```
HYPRE_SStructGridCreate(MPI_COMM_WORLD, ndim, nparts, &grid);
```

# Block-structured grid example:

## Setting up the grid on process 3

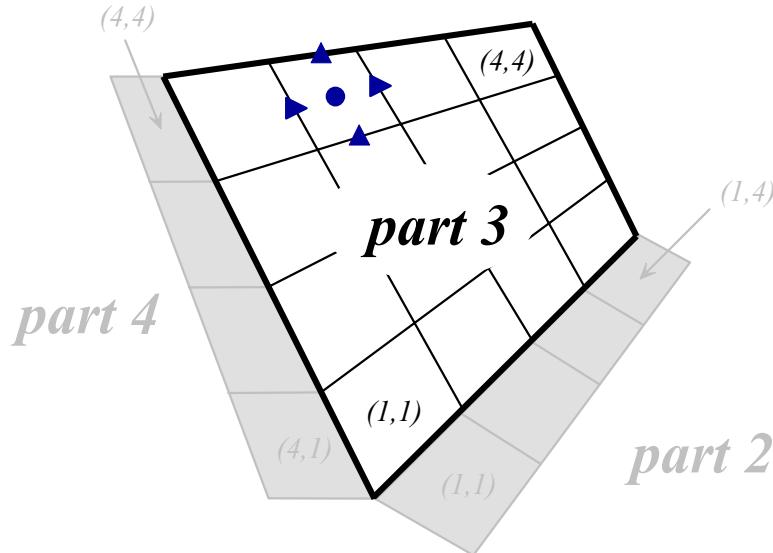


**Set grid extents for  
part 3**

```
int part = 3;  
int ilower[2] = {1,1};  
int iupper[2] = {4,4};  
  
HYPRE_SStructGridSetExtents(grid, part, ilower, iupper);
```

# Block-structured grid example:

## Setting up the grid on process 3



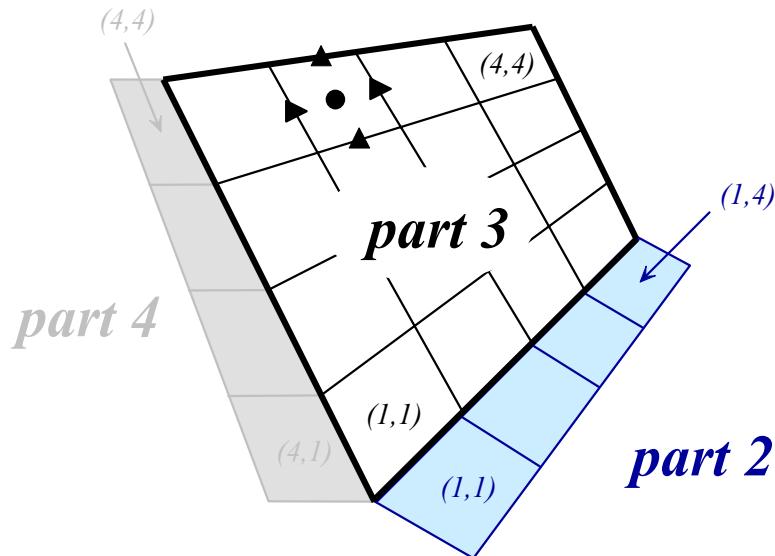
**Set grid variables for  
part 3**

```
int part = 3;
int nvars = 3;
int vartypes[3] = {HYPRE_SSTRUCT_VARIABLE_CELL,
                   HYPRE_SSTRUCT_VARIABLE_XFACE,
                   HYPRE_SSTRUCT_VARIABLE_YFACE};

HYPRE_SStructGridSetVariables(grid, part, nvars, vartypes);
```

# Block-structured grid example:

## Setting up the grid on process 3



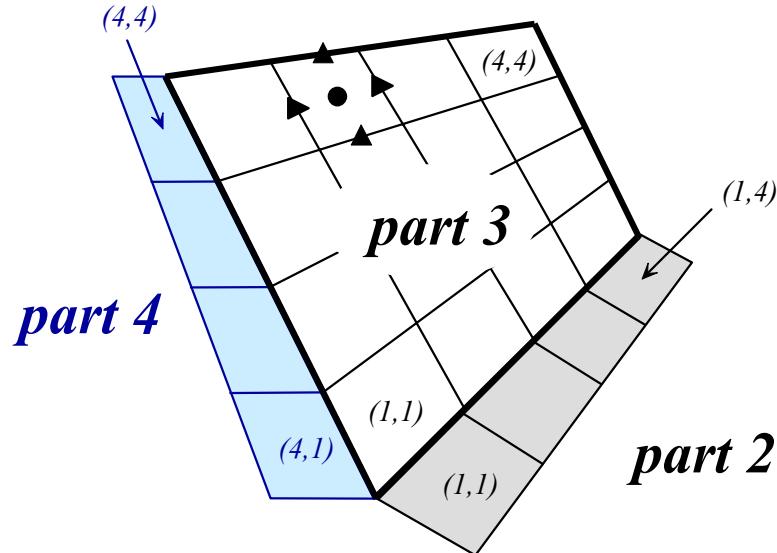
**Set spatial relationship  
between parts 3 and 2**

```
int part = 3, nbor_part = 2;
int ilower[2] = {1,0}, iupper[2] = {4,0};
int nbor_ilower[2] = {1,1}, nbor_iupper[2] = {1,4};
int index_map[2] = {1,0};

HYPRE_SStructGridSetNeighborBox(grid, part, ilower, iupper,
nbor_part, nbor_ilower, nbor_iupper, index_map);
```

# Block-structured grid example:

## Setting up the grid on process 3



Set spatial relationship  
between parts 3 and 4

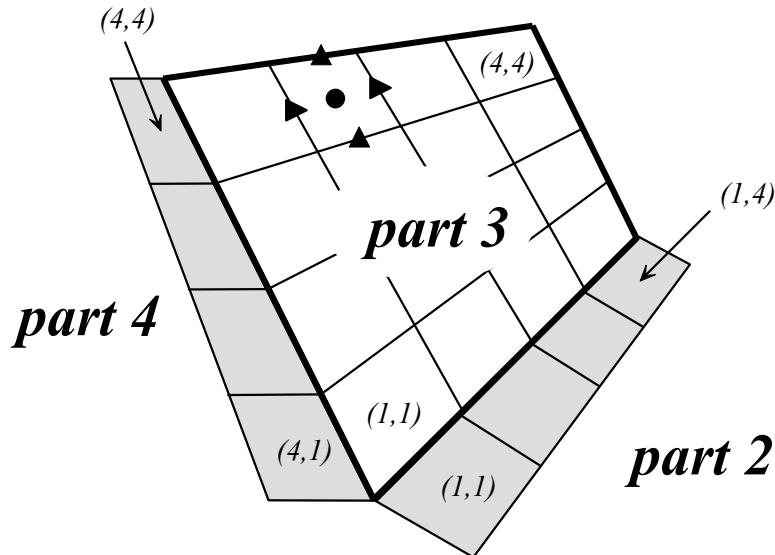
```
int part = 3, nbor_part = 4;
int ilower[2] = {0,1}, iupper[2] = {0,4};
int nbor_ilower[2] = {4,1}, nbor_iupper[2] = {4,4};
int index_map[2] = {0,1};

HYPRE_SStructGridSetNeighborBox(grid, part, ilower, iupper,
nbor_part, nbor_ilower, nbor_iupper, index_map);
```

# Block-structured grid example: Setting up the grid on process 3

---

---

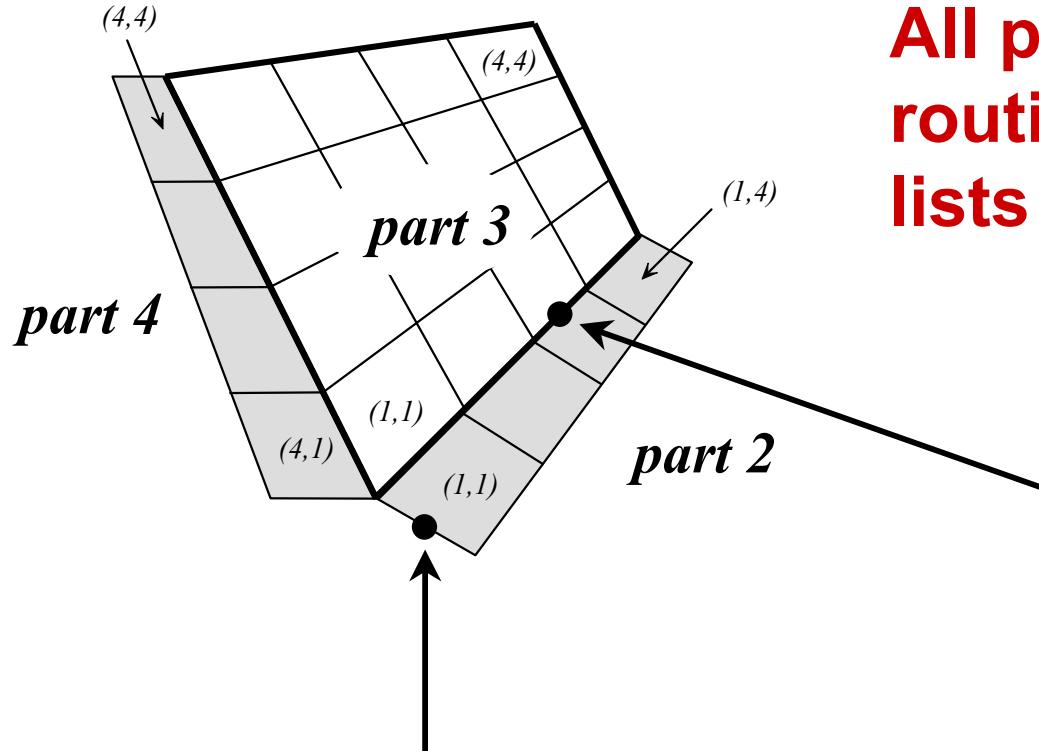


**Assemble the grid**

```
HYPRE_SStructGridAssemble(grid);
```

# Block-structured grid example: some comments on `SetNeighborBox()`

---

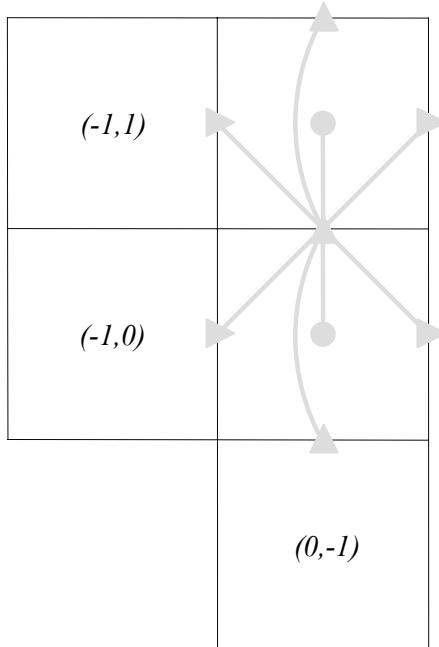


All parts related via this routine must have consistent lists of variables and types

Variables may have different types on different parts (e.g., y-face on part 3 and x-face on part 2)

Some variables on different parts become “the same”

# Block-structured grid example: Setting up the y-face stencil (all processes)



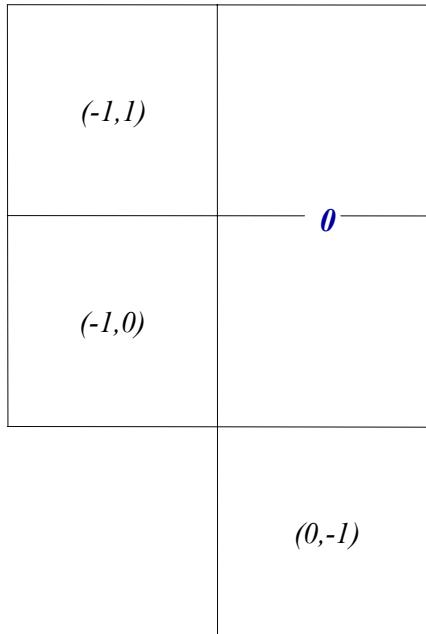
stencil entries	↔	geometries
0	↔	(0,0);▲
1	↔	(0,-1);▲
2	↔	(0,1);▲
3	↔	(0,0);●
4	↔	(0,1);●
5	↔	(-1,0);▶
6	↔	(0,0);▶
7	↔	(-1,1);▶
8	↔	(0,1);▶

Create the stencil object

```
HYPRE_SStructStencil stencil;
int ndim = 2;
int size = 9;

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

# Block-structured grid example: Setting up the y-face stencil (all processes)



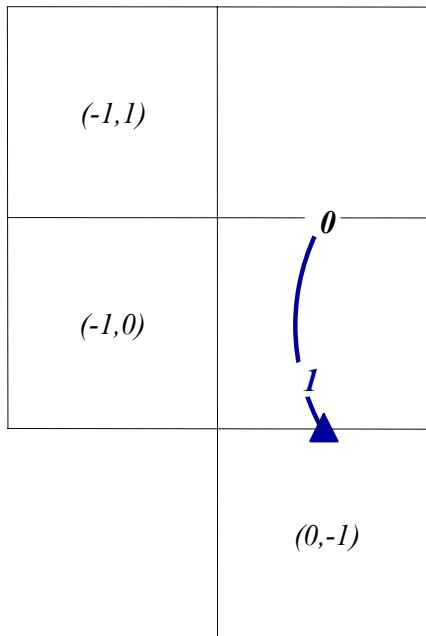
stencil entries	$\leftrightarrow$	geometries
0	$\leftrightarrow$	$(0,0); \blacktriangle$
1	$\leftrightarrow$	$(0,-1); \blacktriangle$
2	$\leftrightarrow$	$(0,1); \blacktriangle$
3	$\leftrightarrow$	$(0,0); \bullet$
4	$\leftrightarrow$	$(0,1); \bullet$
5	$\leftrightarrow$	$(-1,0); \blacktriangleright$
6	$\leftrightarrow$	$(0,0); \blacktriangleright$
7	$\leftrightarrow$	$(-1,1); \blacktriangleright$
8	$\leftrightarrow$	$(0,1); \blacktriangleright$

**Set stencil entries**

```
int entry = 0;
int offset[2] = {0,0};
int var = 2; /* the y-face variable number */

HYPRE_SStructSetStencilEntry(stencil, entry, offset, var);
```

# Block-structured grid example: Setting up the y-face stencil (all processes)



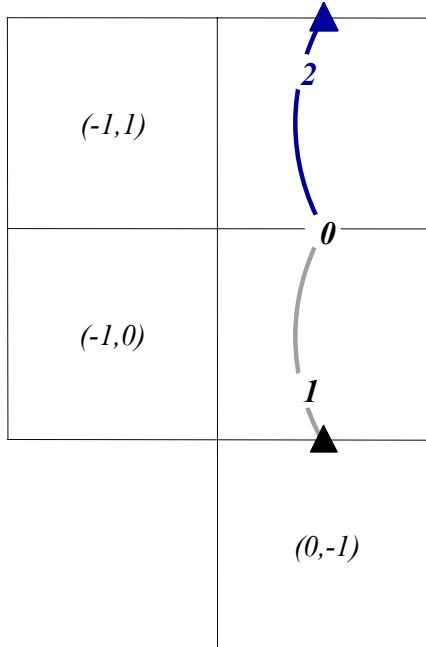
stencil entries	geometries
0	$\leftrightarrow (0,0); \blacktriangle$
1	$\leftrightarrow (0,-1); \blacktriangle$
2	$\leftrightarrow (0,1); \blacktriangle$
3	$\leftrightarrow (0,0); \bullet$
4	$\leftrightarrow (0,1); \bullet$
5	$\leftrightarrow (-1,0); \blacktriangleright$
6	$\leftrightarrow (0,0); \blacktriangleright$
7	$\leftrightarrow (-1,1); \blacktriangleright$
8	$\leftrightarrow (0,1); \blacktriangleright$

**Set stencil entries**

```
int entry = 1;
int offset[2] = {0,-1};
int var = 2; /* the y-face variable number */

HYPRE_SStructSetStencilEntry(stencil, entry, offset, var);
```

# Block-structured grid example: Setting up the y-face stencil (all processes)



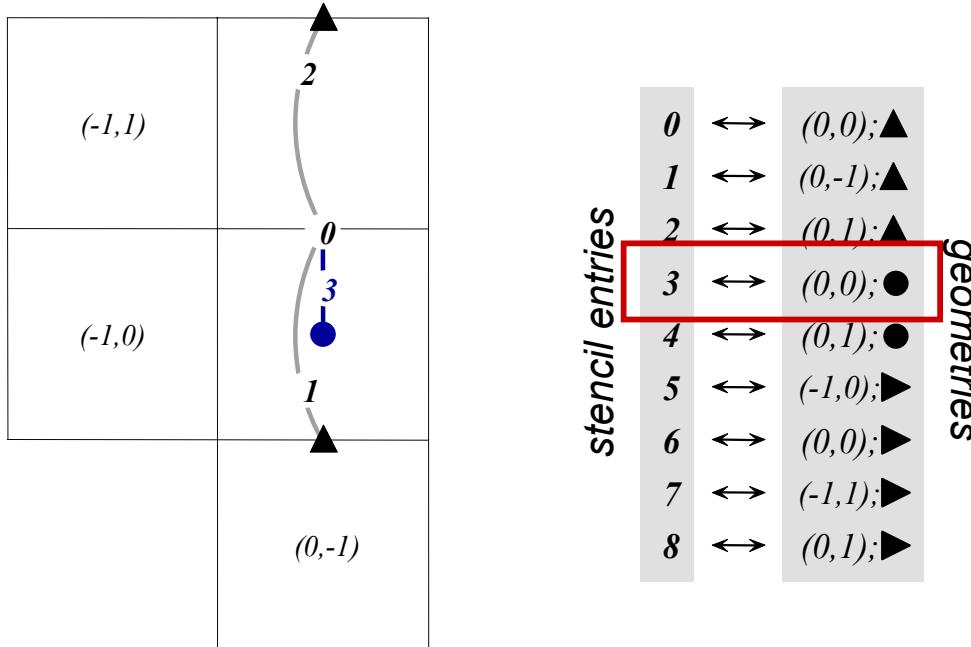
stencil entries	↔	geometries
0	↔	(0,0);▲
1	↔	(0,-1);▲
2	↔	(0,1);▲
3	↔	(0,0);●
4	↔	(0,1);●
5	↔	(-1,0);►
6	↔	(0,0);►
7	↔	(-1,1);►
8	↔	(0,1);►

**Set stencil entries**

```
int entry = 2;
int offset[2] = {0,1};
int var = 2; /* the y-face variable number */
```

```
HYPRE_SStructSetStencilEntry(stencil, entry, offset, var);
```

# Block-structured grid example: Setting up the y-face stencil (all processes)

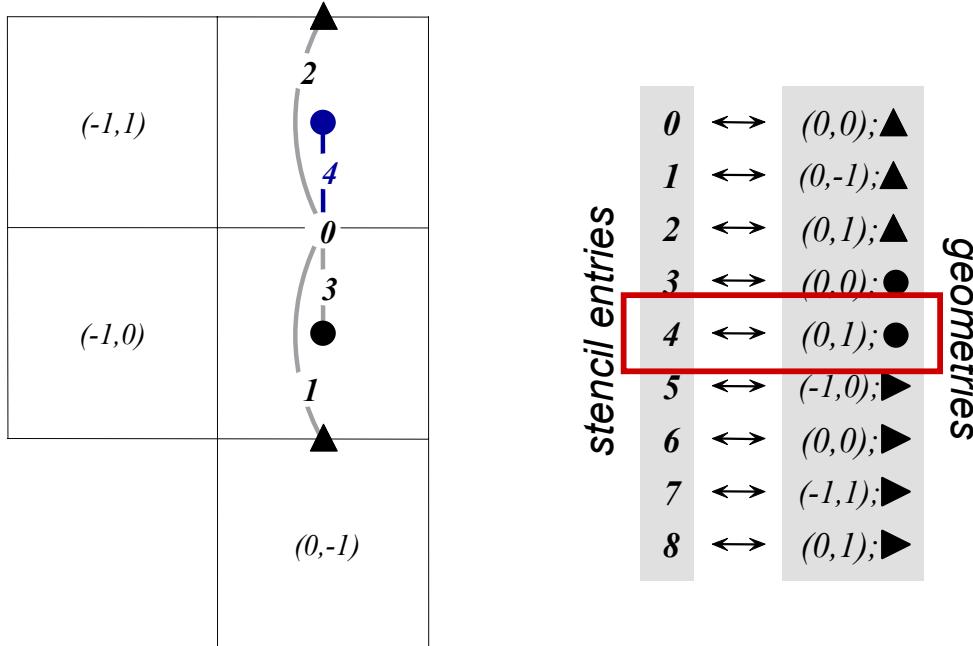


Set stencil entries

```
int entry = 3;
int offset[2] = {0,0};
int var = 0; /* the cell-centered variable number */

HYPRE_SStructSetStencilEntry(stencil, entry, offset, var);
```

# Block-structured grid example: Setting up the y-face stencil (all processes)

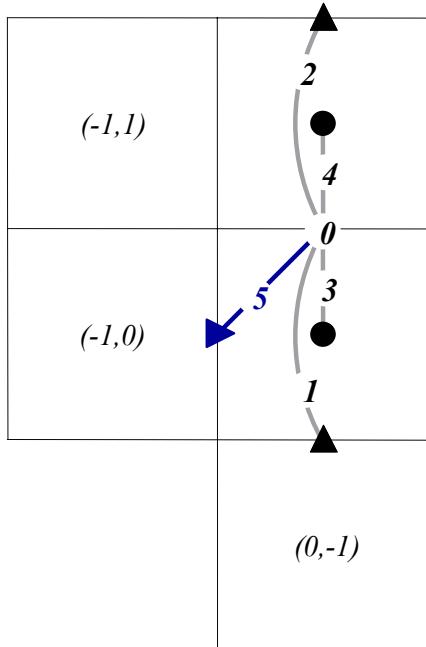


**Set stencil entries**

```
int entry = 4;
int offset[2] = {0,1};
int var = 0; /* the cell-centered variable number */

HYPRE_SStructSetStencilEntry(stencil, entry, offset, var);
```

# Block-structured grid example: Setting up the y-face stencil (all processes)



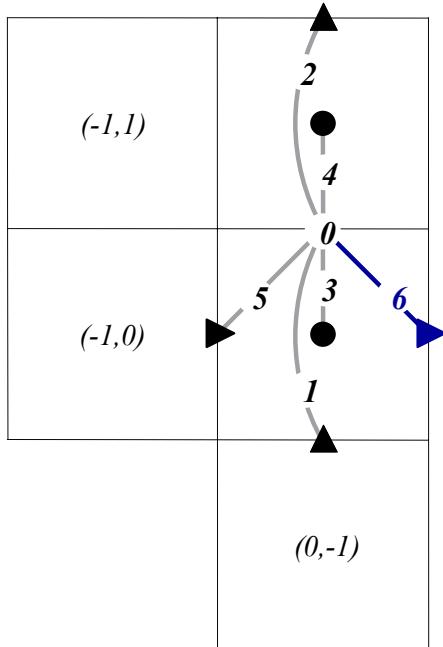
stencil entries	$\leftrightarrow$	geometries
0	$\leftrightarrow$	(0,0);▲
1	$\leftrightarrow$	(0,-1);▲
2	$\leftrightarrow$	(0,1);▲
3	$\leftrightarrow$	(0,0);●
4	$\leftrightarrow$	(0,1);●
5	$\leftrightarrow$	(-1,0);▶
6	$\leftrightarrow$	(0,0);▶
7	$\leftrightarrow$	(-1,1);▶
8	$\leftrightarrow$	(0,1);▶

Set stencil entries

```
int entry = 5;
int offset[2] = {-1,0};
int var = 1; /* the x-face variable number */

HYPRE_SStructSetStencilEntry(stencil, entry, offset, var);
```

# Block-structured grid example: Setting up the y-face stencil (all processes)



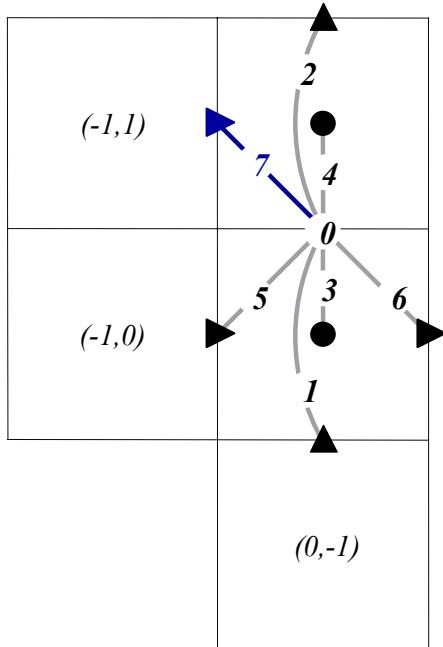
stencil entries	$\leftrightarrow$	geometries
0	$\leftrightarrow$	(0,0);▲
1	$\leftrightarrow$	(0,-1);▲
2	$\leftrightarrow$	(0,1);▲
3	$\leftrightarrow$	(0,0);●
4	$\leftrightarrow$	(0,1);●
5	$\leftrightarrow$	(-1,0);▶
6	$\leftrightarrow$	(0,0);▶
7	$\leftrightarrow$	(-1,1);▶
8	$\leftrightarrow$	(0,1);▶

Set stencil entries

```
int entry = 6;
int offset[2] = {0,0};
int var = 1; /* the x-face variable number */

HYPRE_SStructSetStencilEntry(stencil, entry, offset, var);
```

# Block-structured grid example: Setting up the y-face stencil (all processes)



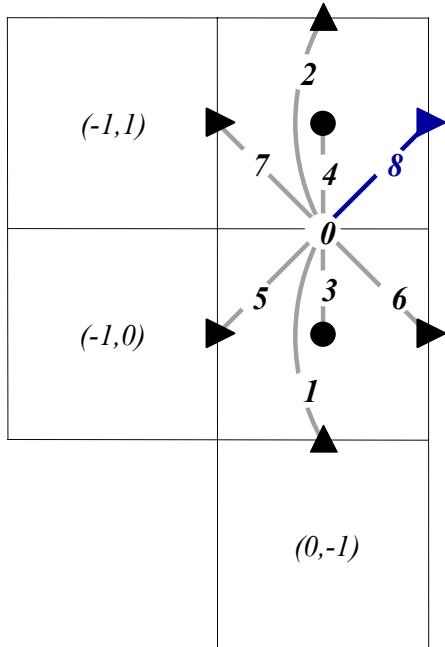
stencil entries	↔	geometries
0	↔	(0,0);▲
1	↔	(0,-1);▲
2	↔	(0,1);▲
3	↔	(0,0);●
4	↔	(0,1);●
5	↔	(-1,0);▶
6	↔	(0,0);▶
7	↔	(-1,1);▶
8	↔	(0,1);▶

**Set stencil entries**

```
int entry = 7;
int offset[2] = {-1,1};
int var = 1; /* the x-face variable number */

HYPRE_SStructSetStencilEntry(stencil, entry, offset, var);
```

# Block-structured grid example: Setting up the y-face stencil (all processes)



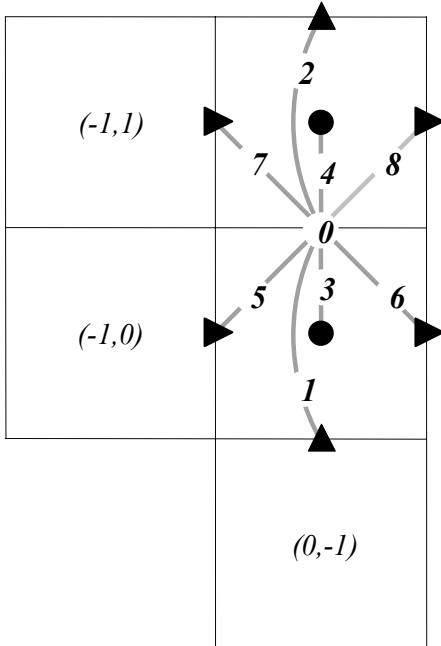
stencil entries	↔	geometries
0	↔	(0,0);▲
1	↔	(0,-1);▲
2	↔	(0,1);▲
3	↔	(0,0);●
4	↔	(0,1);●
5	↔	(-1,0);▶
6	↔	(0,0);▶
7	↔	(-1,1);▶
8	↔	(0,1);▶

Set stencil entries

```
int entry = 8;
int offset[2] = {0,1};
int var = 1; /* the x-face variable number */

HYPRE_SStructSetStencilEntry(stencil, entry, offset, var);
```

# Block-structured grid example: Setting up the y-face stencil (all processes)



stencil entries	$\leftrightarrow$	geometries
0	$\leftrightarrow$	(0,0); ▲
1	$\leftrightarrow$	(0,-1); ▲
2	$\leftrightarrow$	(0,1); ▲
3	$\leftrightarrow$	(0,0); ●
4	$\leftrightarrow$	(0,1); ●
5	$\leftrightarrow$	(-1,0); ▶
6	$\leftrightarrow$	(0,0); ▶
7	$\leftrightarrow$	(-1,1); ▶
8	$\leftrightarrow$	(0,1); ▶

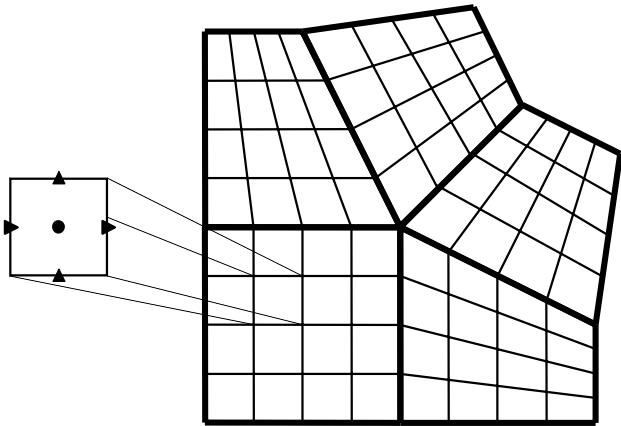
**That's it!**  
**There is no assemble routine**

# Block-structured grid example:

## Setting up the graph on process 3

---

---



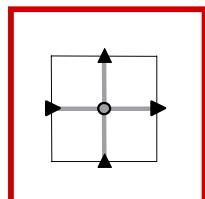
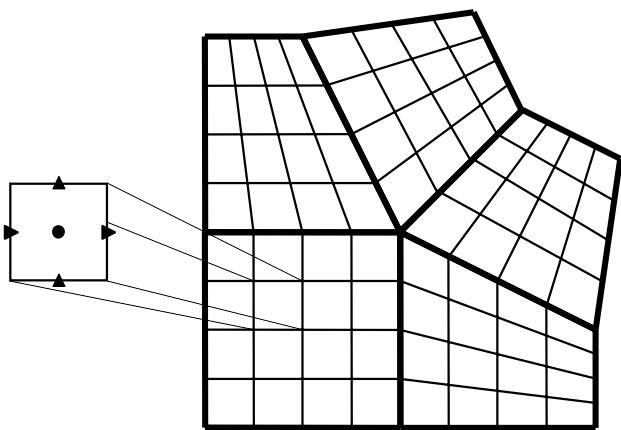
Create the graph  
object

```
HYPRE_SStructGraph graph;
```

```
HYPRE_SStructGraphCreate(MPI_COMM_WORLD, grid, &graph);
```

# Block-structured grid example:

## Setting up the graph on process 3



Set the cell-centered stencil for each part

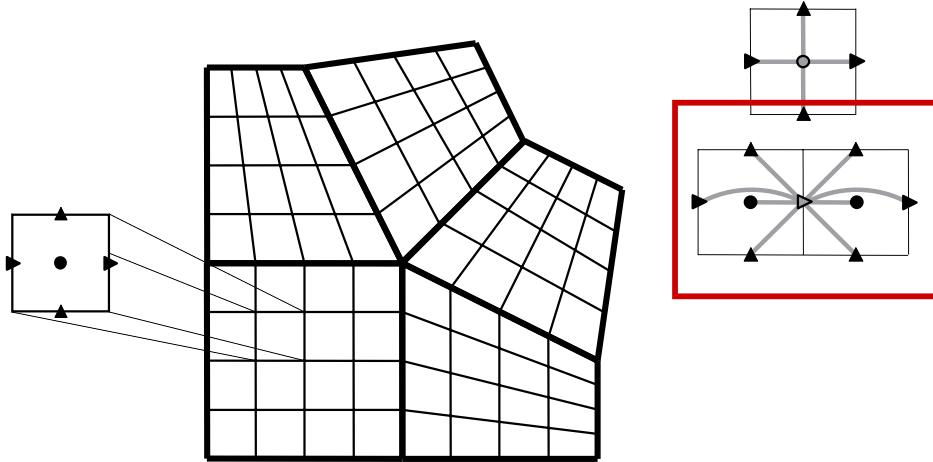
```
int part;
int var = 0;
HYPRE_SStructStencil cell_stencil;

HYPRE_SStructGraphSetStencil(graph, part, var, cell_stencil);
```

# Block-structured grid example: Setting up the graph on process 3

---

---



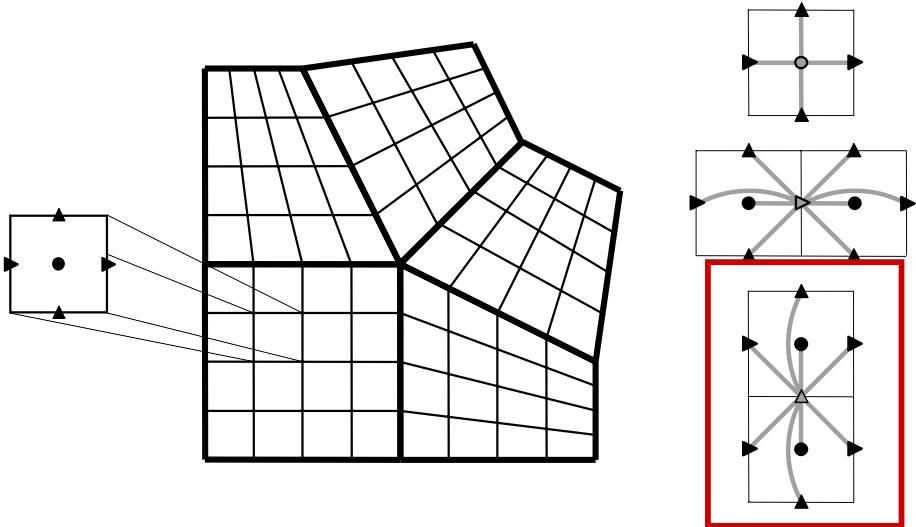
**Set the x-face stencil  
for each part**

```
int part;
int var = 1;
HYPRE_SStructStencil x_stencil;

HYPRE_SStructGraphSetStencil(graph, part, var, x_stencil);
```

# Block-structured grid example: Setting up the graph on process 3

---



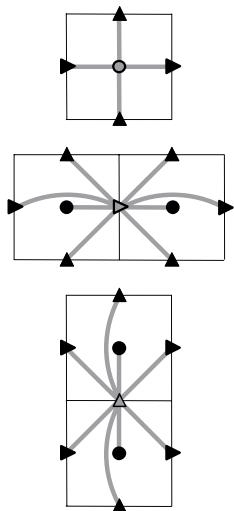
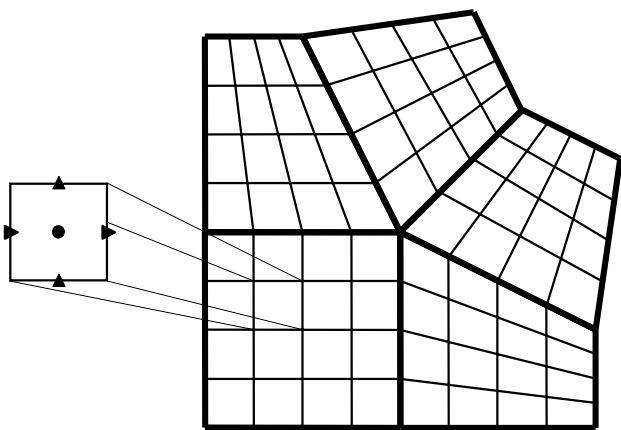
**Set the y-face stencil  
for each part**

```
int part;
int var = 2;
HYPRE_SStructStencil y_stencil;

HYPRE_SStructGraphSetStencil(graph, part, var, y_stencil);
```

# Block-structured grid example: Setting up the graph on process 3

---



**Assemble the graph**

```
/* No need to add non-stencil entries  
 * with HYPRE_SStructGraphAddEntries */  
  
HYPRE_SStructGraphAssemble(graph);
```

# Block-structured grid example:

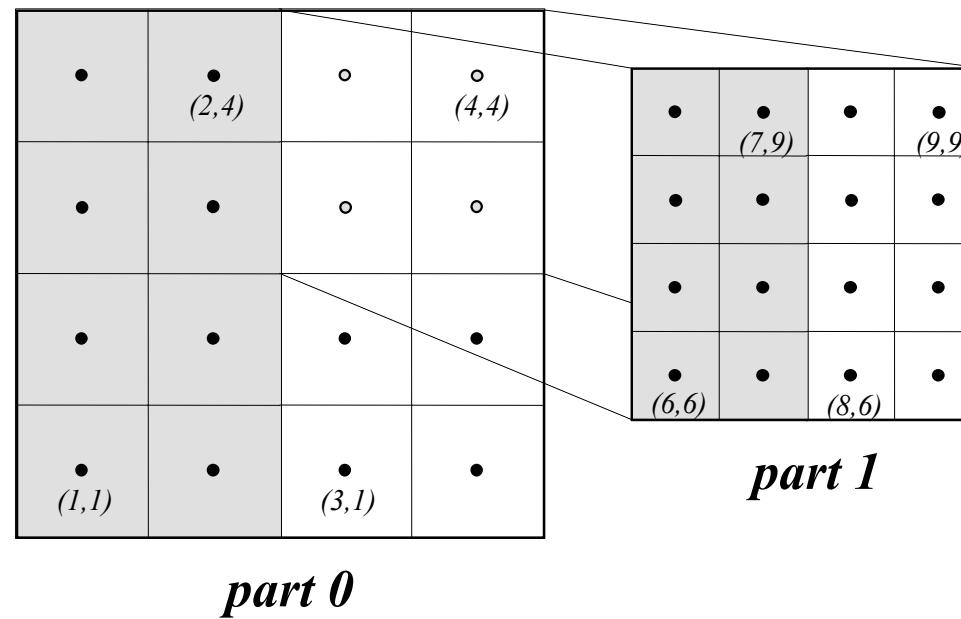
## Setting up the matrix and vector

---

- The matrix and vector objects are constructed in a manner similar to the Struct interface
- Matrix coefficients are set with the routines
  - `HYPRE_SStructMatrixSetValues`
  - `HYPRE_SStructMatrixAddToValues`
- Vector values are set with similar routines
  - `HYPRE_SStructVectorSetValues`
  - `HYPRE_SStructVectorAddToValues`

# Structured AMR example (SStruct)

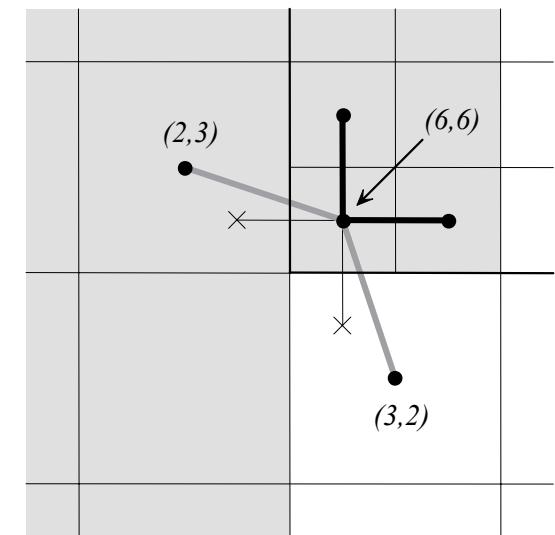
- Consider a simple cell-centered discretization of the Laplacian on the following structured AMR grid



- Each AMR grid level is defined as a separate part
- Assume 2 processes with shaded regions on process 0 and unshaded regions on process 1

# Structured AMR example (SStruct)

- The **grid** is constructed using straightforward calls to the routines `HYPRE_SStructGridSetExtents` and `HYPRE_SStructGridSetVariables` as in the previous block-structured grid example
- The **graph** is constructed from a cell-centered stencil plus additional *non-stencil entries* at coarse-fine interfaces
- These non-stencil entries are set one variable at a time using `HYPRE_SStructGraphAddEntries()`



# 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)

# Comments on SStruct interface

---

- The routine `HYPRE_SStructGridAddVariables` allows additional variables to be added to individual cells, but is not yet implemented
- The routine `HYPRE_SStructGridSetNeighborBox` only supports cell-centered variable types currently
- Implementing an FAC solver in *hypre* for AMR
- Working with AMR users to evaluate and improve the SStruct interface for SAMR applications
- Will release a *hypre*-independent specification and implementation of SStruct to enable access to solvers in other libraries such as PETSc
- Will work with CCA to “componentize” SStruct

# Finite Element Interface (FEI)

---

---

- The FEI interface is designed for finite element discretizations on unstructured grids
- See the following for information on how to use it

R. L. Clay et al. An annotated reference guide to the Finite Element Interface (FEI) Specification, Version 1.0. Sandia National Laboratories, Livermore, CA, Technical Report SAND99-8229, 1999.

# Linear-Algebraic System Interface (IJ)

---

- The IJ interface provides access to general sparse-matrix solvers, but not specialized solvers
- There are two basic steps involved:
  - set up the Matrix
  - set up the right-hand-side Vector
- Consider a 5pt 2D Laplacian problem on a 10x10 grid

$$\begin{pmatrix} A & -I & & & \\ -I & A & \ddots & & \\ & \ddots & \ddots & -I & \\ & & \ddots & \ddots & -I \\ & & & -I & A \end{pmatrix}, \quad A = \begin{pmatrix} 4 & -1 & & & \\ -1 & 4 & \ddots & & \\ & \ddots & \ddots & -1 & \\ & & \ddots & \ddots & -1 \\ & & & -1 & 4 \end{pmatrix}$$

# An example for the IJ interface: *setting up the Matrix*

---

```
HYPRE_IJMatrix A;
HYPRE_ParCSRMatrix parcsr_A;
int nrows = 3, 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,-1,-1, -1, 4, -1, -1, -1, 4, -1,-1};

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, &parcsr_A);
```

# An example for the IJ interface: *setting up the right-hand-side Vector*

---

```
HYPRE_IJVector b;
HYPRE_ParVector par_b;
int jlower, jupper, nvalues = 100;
int indices[100] = {0,1,...,99};
double values[100] = {1,1,...,1};

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, &par_b);
```

# Additional IJ interface function calls

---

IJMatrix function calls:

```
HYPRE_IJMatrixAddToValues(A, nrows, ncols, rows, cols, values)
```

```
HYPRE_IJMatrixGetValues(A, nrows, ncols, rows, cols, values)
```

For better efficiency:

```
HYPRE_IJMatrixSetRowSizes(A, sizes)
```

```
HYPRE_IJMatrixSetDiagOffdSizes(A, diag_sizes, offdiag_sizes)
```

IJVector function calls:

```
HYPRE_IJVectorAddToValues(b, nvalues, indices, values)
```

```
HYPRE_IJVectorGetValues(b, nvalues, indices, values)
```

# Several Solvers and Preconditioners are available in *hypre*

---

- Current solver availability via conceptual interfaces

Solvers	System Interfaces			
	Struct	SStruct	FEI	IJ
Jacobi	X			
SMG	X			
PFMG	X			
BoomerAMG	X	X	X	X
ParaSails	X	X	X	X
PILUT	X	X	X	X
Euclid	X	X	X	X
PCG	X	X	X	X
GMRES	X	X	X	X

# 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 conceptual interface**

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

- **Destroy the solver**

```
HYPRE_SolverDestroy(solver);
```

# Solver example: SMG-PCG

---

```
HYPRE_StructPCGCreate(MPI_COMM_WORLD, &solver);

/* set stopping criteria */
HYPRE_StructPCGSetTol(solver, 1.0e-06);

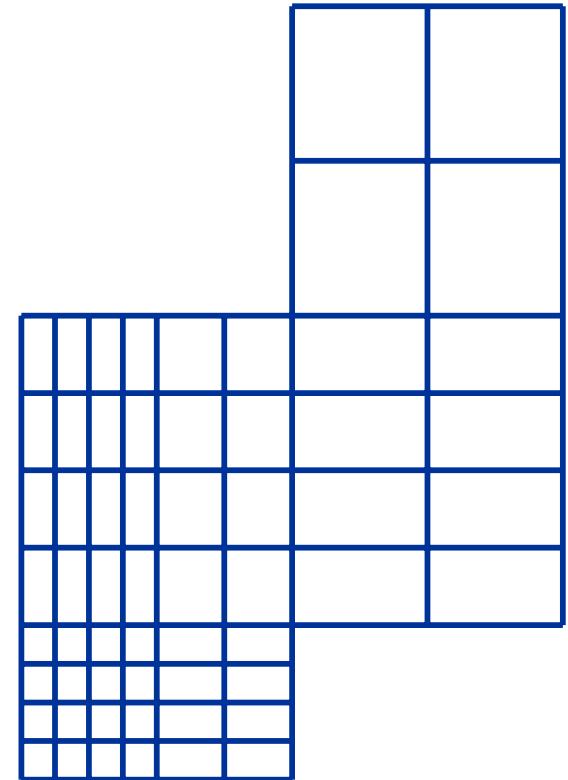
/* define preconditioner (one symmetric V(1,1)-cycle) */
HYPRE_StructSMGCreate(MPI_COMM_WORLD, &precond);
HYPRE_StructSMGSetMaxIter(precond, 1);
HYPRE_StructSMGSetTol(precond, 0.0);
HYPRE_StructSMGSetZeroGuess(precond);
HYPRE_StructSMGSetNumPreRelax(precond, 1);
HYPRE_StructSMGSetNumPostRelax(precond,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

/* set preconditioner and solve */
HYPRE_StructPCGSetPrecond(solver,
    HYPRE_StructSMGSolve, HYPRE_StructSMGSetup, precond);
HYPRE_StructPCGSetup(solver, A, b, x);
HYPRE_StructPCGSolve(solver, A, b, x);
```

# SMG and PFMG are semicoarsening multigrid methods for structured grids

---

- Interface: Struct
  - Matrix Class: Struct
- 
- SMG uses plane smoothing in 3D, where each plane “solve” is effected by one 2D V-cycle
  - SMG is very robust
  - PFMG uses simple pointwise smoothing, and is less robust

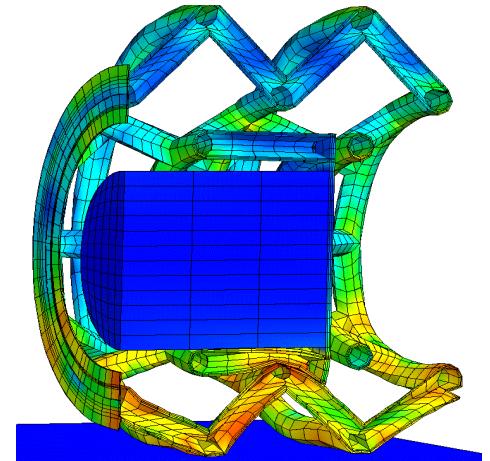


*Our largest run (with PFMG-CG): 1B unknowns on 3150 processors of ASCI Red in 54 seconds*

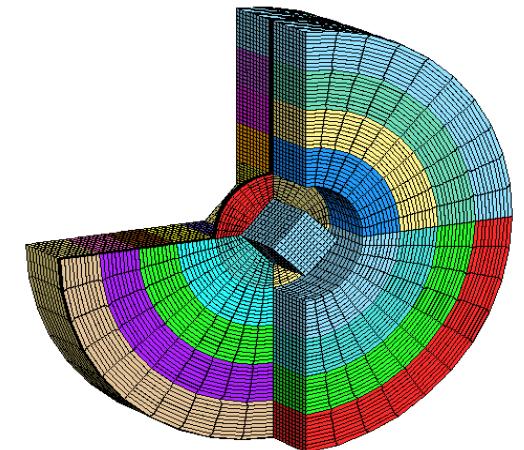
# BoomerAMG is an algebraic multigrid method for unstructured grids

---

- Interface: SStruct, FEI, IJ
  - Matrix Class: ParCSR
- 
- Originally developed as a general matrix method (i.e., assumed given A, x, and b only)
  - Uses simple pointwise relaxation
  - Automatically defines coarse “grids”



DYNA3D



PMESH

# BoomerAMG interface functions

---

## Coarsening techniques:

- CLJP coarsening
- various variants of the classical Ruge-Stüben (RS) coarsening
- Falgout coarsening (default)

`HYPRE_BoomerAMGSetCoarsenType(solver, coarsen_type)`

## Relaxation techniques:

- weighted Jacobi relaxation
- hybrid Gauß-Seidel / Jacobi relaxation (default)
- symmetric hybrid Gauß-Seidel / Jacobi relaxation
- further techniques underway

`HYPRE_BoomerAMGSetGridRelaxType(solver, relax_type)`

`HYPRE_BoomerAMGSetGridRelaxPoints(solver, relax_pts)`

# More BoomerAMG interface functions

---

`HYPRE_BoomerAMGSetMaxLevels(solver, max_levels)`

**default:** 25

`HYPRE_BoomerAMGSetMaxIter(solver, max_iter)`

**default:** 20

`HYPRE_BoomerAMGSetTol(solver, tol)`

**default:**  $10^{-7}$

`HYPRE_BoomerAMGSetStrongThreshold(solver, strong_thr)`

**default:** 0.25

`HYPRE_BoomerAMGSetMaxRowSum(solver, max_row_sum)`

**for more efficient treatment of very diagonally dominant portions of the matrix, default: 0.9**

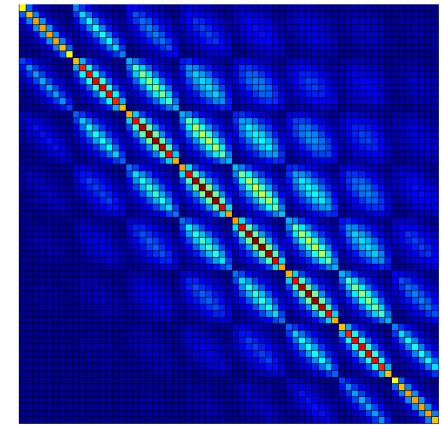
`HYPRE_BoomerAMGSetNumGridSweeps(solver, num_grid_sweeps)`

# ParaSAILS is an approximate inverse method for sparse linear systems

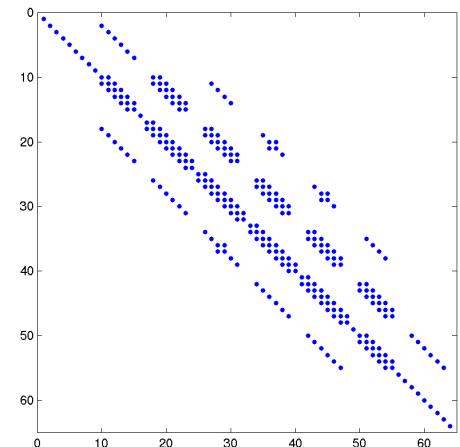
---

- Interface: SStruct, FEI, IJ
  - Matrix Class: ParCSR
- 
- Approximates the inverse of  $A$  by a sparse matrix  $M$  by minimizing the Frobenius norm of  $I - AM$
  - Uses graph theory to predict good sparsity patterns for  $M$

Exact inverse



Approx inverse



# ParaSails interface functions

---

```
HYPRE_ParaSailsCreate(MPI_Comm comm, HYPRE_Solver  
*solver, int symmetry)
```

- 0 nonsymm./indefinite problem, nonsymm. Preconditioner (default)
- 1 SPD problem, SPD (factored) preconditioner
- 2 nonsymm./definite problem, SPD (factored) preconditioner

```
HYPRE_ParaSailsSetParams(HYPRE_Solver solver, double  
thresh, int nlevel, double filter)
```

thresh: drops all entries in symm. diagonally scaled A less than  
thresh, default: 0.1

nlevel: m = nlevel+1 , default: 1

filter: post-thresholding procedure, default: 0.1

```
HYPRE_ParaSailsSetLoadbal(solver, double loadbal)
```

```
HYPRE_ParaSailsSetReuse(solver, int reuse)
```

# PILUT is an Incomplete LU method for sparse linear systems

---

- Interface: SStruct, FEI, IJ
  - Matrix Class: ParCSR
- 
- Uses thresholding drop strategy plus a mechanism to control the maximum size of the ILU factors
  - originally by Kumar and Karypis for T3D
  - now uses MPI and more coarse-grain parallelism
  - uses Schur-complement approach to parallelism

# PILUT interface functions

---

---

`HYPRE_ParCSRilutSetDropTolerance (solver, tol)`

**default:** 0.0001

`HYPRE_ParCSRilutSetFactorRowSize (solver, size)`

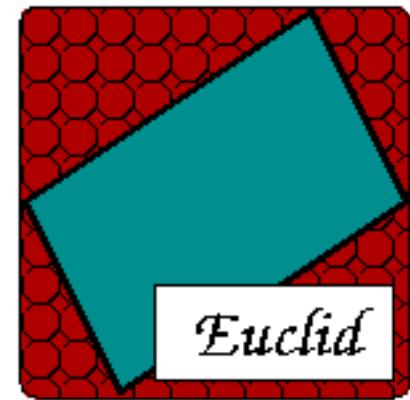
**default:** 20

# Euclid is a family of Incomplete LU methods for sparse linear systems

---

---

- **Interface:** SStruct, FEI, IJ
  - **Matrix Class:** ParCSR
- 
- **Obtains scalable parallelism via local and global reorderings**
  - **Good for unstructured problems**
- 
- <http://www.cs.odu.edu/~hysom/Euclid>



# Euclid interface functions

---

`HYPRE_EuclidSetParams(solver, argc, argv)`

if passed on command line

or

`HYPRE_EuclidSetParam(solver, name, value)`

for each parameter

## Options:

- level** factorization level for ILU(k)
- bj** use block Jacobi preconditioning instead of PILU
- eu\_stats** prints summary of timing information
- eu\_mem** prints summary of memory usage

# Krylov solvers interface functions

---

```
HYPRE_ParCSRPCGSetMaxIter(solver, max_iter)
HYPRE_ParCSRPCGSetTol(solver, tol)
HYPRE_ParCSRPCGGetNumIterations(solver, iter)
HYPRE_ParCSRPCGGetFinalRelativeResidualNorm(solver, norm)

HYPRE_ParCSRGMRESSetKDim(solver, k_dim)
HYPRE_ParCSRGMRESSetMaxIter(solver, max_iter)
HYPRE_ParCSRGMRESSetTol(solver, tol)
HYPRE_ParCSRGMRESGetNumIterations(solver, iter)
HYPRE_ParCSRGMRESGetFinalRelativeResidualNorm(solver,
                                              norm)
```

# Getting the code

---

---

<http://www.llnl.gov/CASC/hypre/>



- The User's Manual and Reference Manual can be downloaded directly
- A form must be filled out prior to download. This is just for our own records.

# Building the library

---

- Usually, *hypre* can be built by typing `configure` followed by `make`
- Configure supports several options (for usage information, type '`configure --help`'):
  - '`configure --enable-debug`' - turn on debugging
  - '`configure --with-openmp`' - use openmp
  - '`configure --with-CFLAGS=...`' - set compiler flags

# Calling *hypre* from Fortran

---

- C code:

```
HYPRE_IJMatrix A;  
int nvalues, row, *cols;  
double *values;  
  
HYPRE_IJMatrixSetValues(A, nvalues, row, cols, values);
```

- Corresponding Fortran code:

```
integer*8 A  
integer nvalues, row, cols(MAX_NVALUES)  
double precision values(MAX_NVALUES)  
  
call HYPRE_IJMatrixSetValues(A, nvalues, row, cols, values)
```

# In the future, language interoperability in *hypre* will be provided via **Babel**

---

- Babel is a language interoperability project at LLNL
- *hypre* served as a prototype code during the initial design and development of Babel
- Hope to include a “Babelized” version of the IJ interface and some ParCSR-based solvers in the next general release of *hypre* (late October?)
- Users will download, build, and use *hypre* in virtually the same manner as before
- Additional language support (e.g., Java) will be available via the **Alexandria** web site

# Reporting bugs

---

---

**<http://www-casc.llnl.gov/bugs>**

- Submit bugs, desired features, and documentation problems, or query the status of previous reports
- First time users must click on “Open a new Bugzilla account” under “User login account management”

---

---

This work was performed under the auspices of the U.S.  
Department of Energy by Lawrence Livermore National  
Laboratory under contract no. W-7405-Eng-48.