UCRL-PRES-202207

An Introduction to the SAMRAI Framework: Parts I-IV

Rich Hornung

Center for Applied Scientific Computing
Lawrence Livermore National Laboratory

www.llnl.gov/CASC/SAMRAI
samrai@llnl.gov

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

Center for Applied Scientific Computing

Topics covered in Parts I-IV

- Part I
  - Basic structured AMR (SAMR) concepts
  - SAMRAI motivation and design goals
  - Summary of SAMRAI library organization
- Part II
  - SAMRAI “hierarchy” classes
    - index space
    - box
    - patch
    - level
    - patch hierarchy

Topics covered in Parts I-IV

- Part II ctd…
  - SAMRAI “variable” and “patchdata” classes
    - cell data
    - node data
    - side data
    - face data
    - edge data
    - index data
- Part III
  - SAMRAI “Variable Database” motivation & usage
- Part IV
  - SAMRAI data communcation infrastructure
    - design motivation and key concepts
    - Refine Algorithm and Refine Schedule
    - Coarsen Algorithm and Coarsen Schedule

Part I

Topics covered in Part I

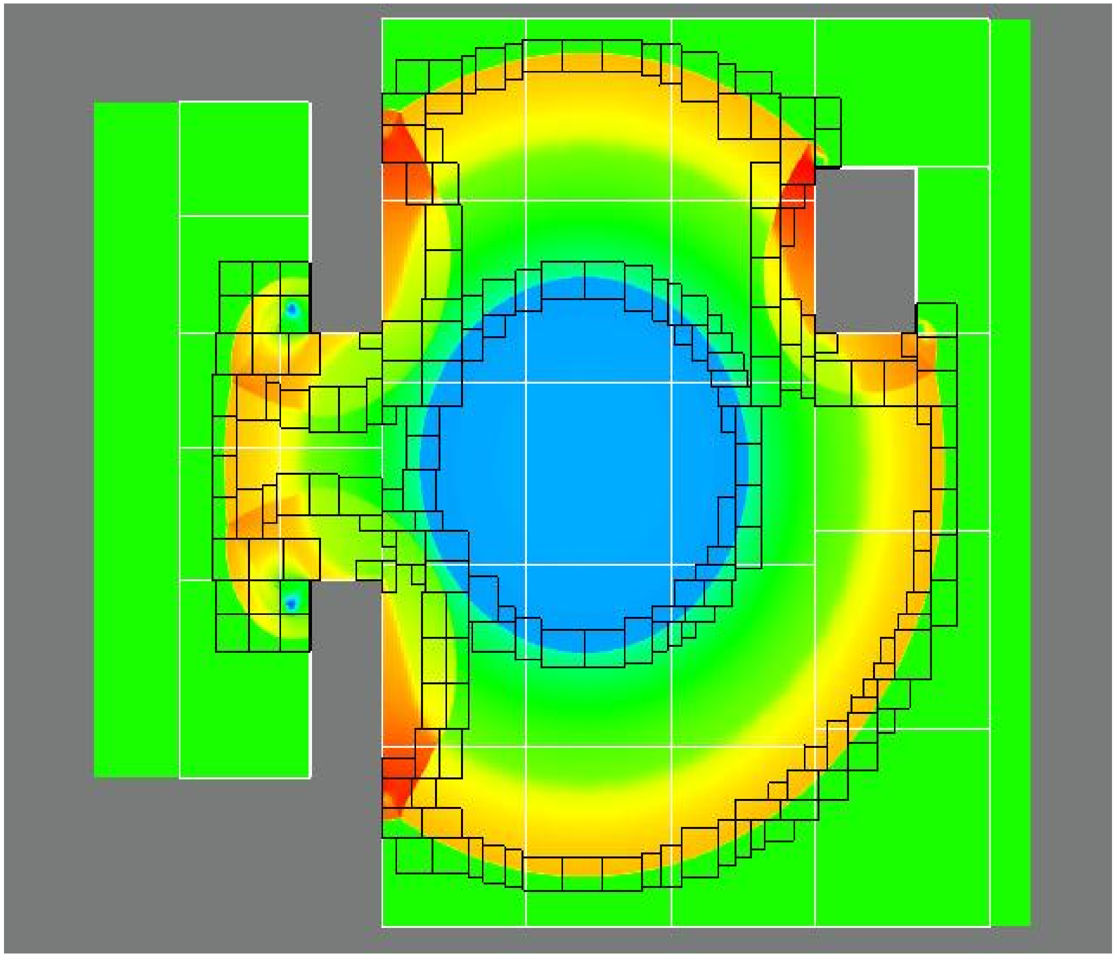
- Basic structured AMR (SAMR) concepts
- SAMRAI motivation and design goals
- Summary of SAMRAI library organization

Basic structure AMR (SAMR) concepts

SAMR employs a dynamic structured "patch hierarchy"

Mesh and data:

- data (e.g., arrays) mapped to "logically-rectangular" patches
- any mesh system mapping to logically-rectangular index space can be used (e.g., Cartesian coords, cylindrical coords, general hexahedra, etc.)

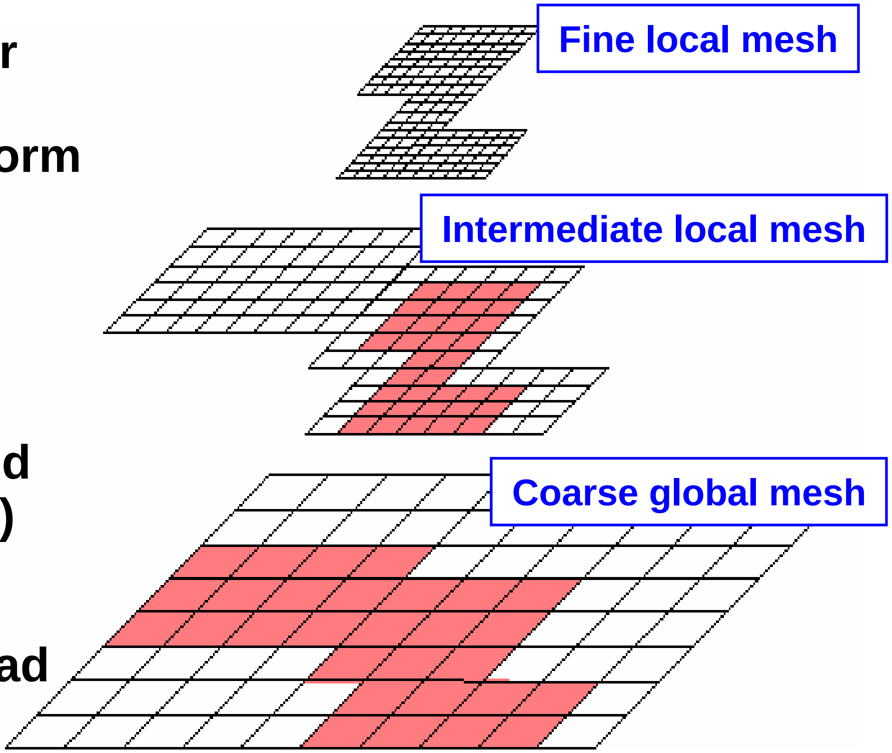


Basic SAMR ingredients:

- problem formulation for locally-refined meshes
- (serial) numerical routines for individual patches
- inter-patch data transfer operations (copying, coarsening, refining, ...)

Structure of SAMR computational mesh

- Hierarchy of levels of mesh resolution
- Finer levels are nested within coarser
- Cells on each level are clustered to form logically-rectangular patches
- Motivation:
  - low overhead mesh description
  - bookkeeping for computation and communication is simple (boxes)
  - simple model of data locality
  - amortize communication overhead by computing over a patch
  - well-suited to structured solvers, hierarchical methods, local time refinement, etc.



Center for Applied Scientific Computing

Structured mesh hierarchy defined using "index spaces"

- Each finer level relates to a coarser level by a "refinement ratio"

Coarsest Level

global index space …(0,0) - (4,3)

patch ………………..(0,0) - (4,3)

Refinement ratio = (4,2)

Intermediate Level

global index space … (0,0) - (19,7)

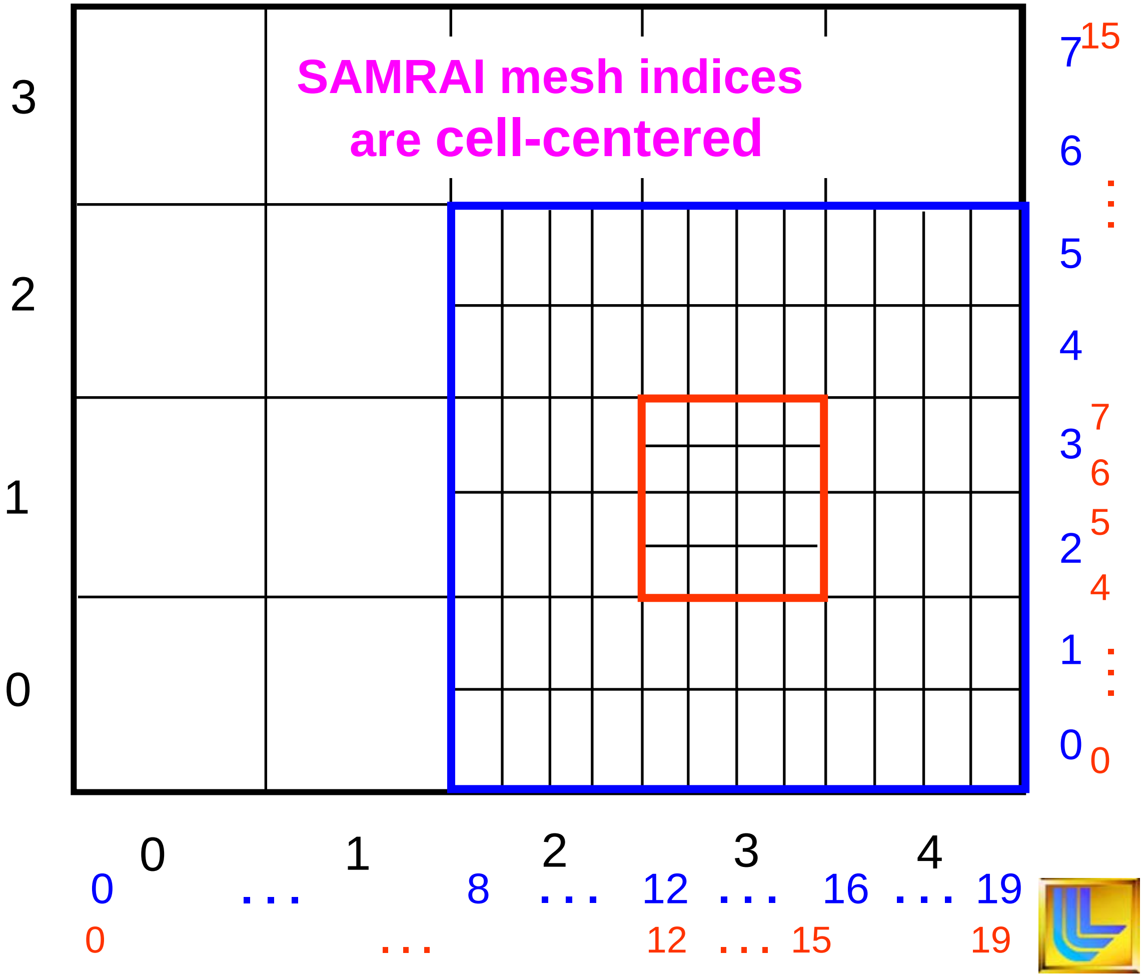
patch ………………..(8,0) - (19,5)

Refinement ratio = (1,2)

Finest Level

global index space… (0,0) - (19,15)

patch ……..……….. (12,4) - (15,7)

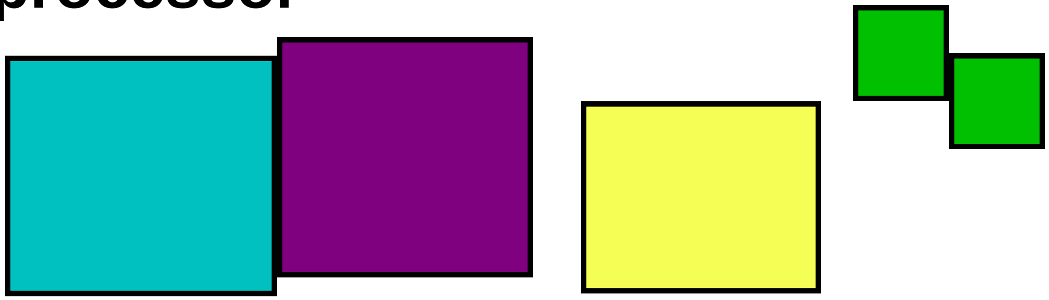


SAMRAI mesh indices are cell-centered

Center for Applied Scientific Computing

SAMRAI decomposes each hierarchy level in parallel individually

- General observations about SAMR applications
  - most parallelism is found at a single mesh level
  - serial numerical operations can be performed on each after communication of necessary boundary data
- SAMRAI assigns each patch and all of its data to one processor



Processor 0
Processor 1
Processor 2
Processor 3

```
exchange patch boundary data
forall patches i in level X
   call user_update(X(i))
end forall
```

parallel data communication

"parallel" loop

serial code

SAMRAI motivation and design goals

SAMRAI: Structured Adaptive Mesh Refinement Application Infrastructure

- **SAMRAI is an object-oriented (C++) software framework for adaptive multi-physics applications**
- **Application folks want to do certain things easily:**
  - quickly focus on numerical methods and solution algorithms
  - build numerical algorithms and coordinate variable data between coupled numerical models
  - easily manipulate data on dynamically changing, locally-refined mesh (data copying, coarsening, refining, time interpolation, …)
- **Main SAMRAI goals: simplify development and management of SAMR applications and enable new algorithm research**

SAMRAI evolves with understanding of application and numerical issues

- **SAMRAI research and development focus:**
  - multi-scale applications, algorithms, and numerical methods
  - adaptive algorithms on massively parallel computing platforms
  - modern software approaches for complicated numerical codes
- **SAMRAI software design goals:**
  - robust code base shared by diverse, complex applications (“infrastructure” common across apps. factored into framework)
  - flexible algorithmic framework to explore new solution methods
  - extensible parallel support for general dynamic data configurations

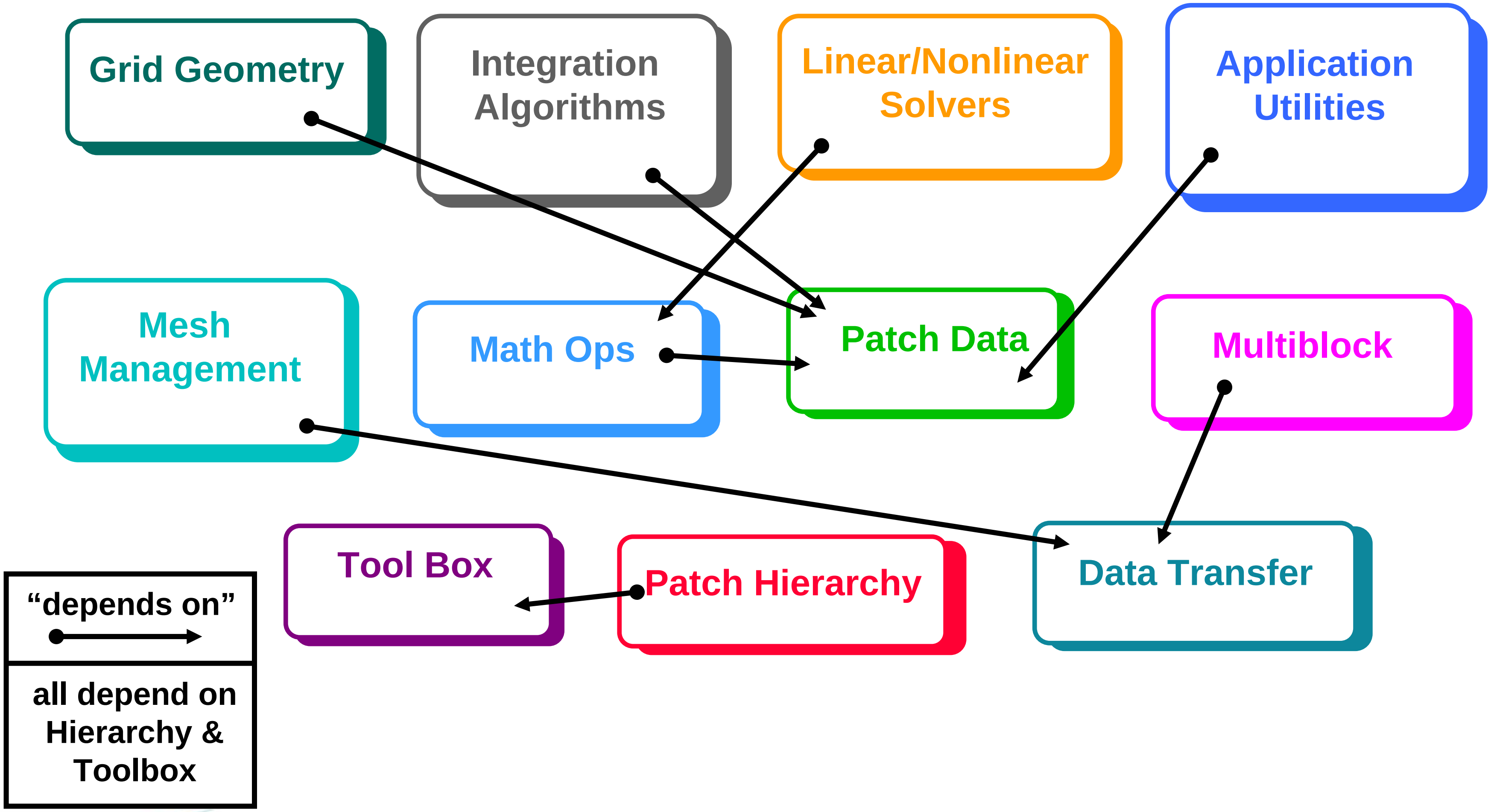
    (extensity *without recompilation*; e.g. via inheritance)

For information about SAMRAI research and application development, visit our web page at www.llnl.gov/CASC/SAMRAI/

Center for Applied Scientific Computing

SAMRAI library organization

User view of SAMRAI is a “toolbox” of classes for application development



Grid Geometry

Integration Algorithms

Linear/Nonlinear Solvers

Application Utilities

Mesh Management

Math Ops

Patch Data

Multiblock

Tool Box

Patch Hierarchy

Data Transfer

“depends on”

all depend on Hierarchy & Toolbox

Summary of SAMRAI packages I

- ***Toolbox*** **--** basic, general utilities used throughout library
  - **smart pointers/containers; memory arenas; MPI classes; input & restart tools; event logging, tracing, statistics, timers**
- ***Hierarchy*** **--** abstract index spaces and patch hierarchy objects
  - **index; box & box containers; patch, patch level, patch hierarchy; interfaces for variable, patch data types; variable database**
- ***Transfer*** **--** inter-patch data movement
  - **communication algorithms & schedules, spatial refine/coarsen and time interpolation operators**
- ***PatchData*** **--** concrete patch data types
  - **variable & patch data classes for array-based data (cell-centered, node-centered, side-centered, …) and data defined on irregular "index" sets**
- ***Math Ops*** **--** basic arithmetic and other operations needed for vector kernels (norms, dot product, etc.)
  - **operationss apply to single patch, single level, or entire hierarchy**

Summary of SAMRAI packages II

- ***Mesh*** **--** adaptive meshing & patch hierarchy construction support
  - **AMR hierarchy contruction/regridding; load balancing**
- **Multiblock –** support data on multiple patch hierarchies
  - **data management and communication between different index spaces**
- ***Algorithm*** **--** solution algorithms for certain PDE problems
  - **local time stepping; method of lines; hyperbolic conservation laws; basic implicit time integration support**
- ***Solvers*** **--** support for linear and nonlinear solvers
  - **vector classes; interfaces and wrappers for PETSc, KINSOL, PVODE; AMR Poisson solver (using *hypre*)**
- ***Geometry*** **--** support for coordinate systems on AMR hierarchy
  - **grid geometry; patch geometry; spatial refine/coarsen operators**
- ***App Utils*** **–** utilities helpful in application construction
  - **simple boundary conditions; visualization file generation**

Part II

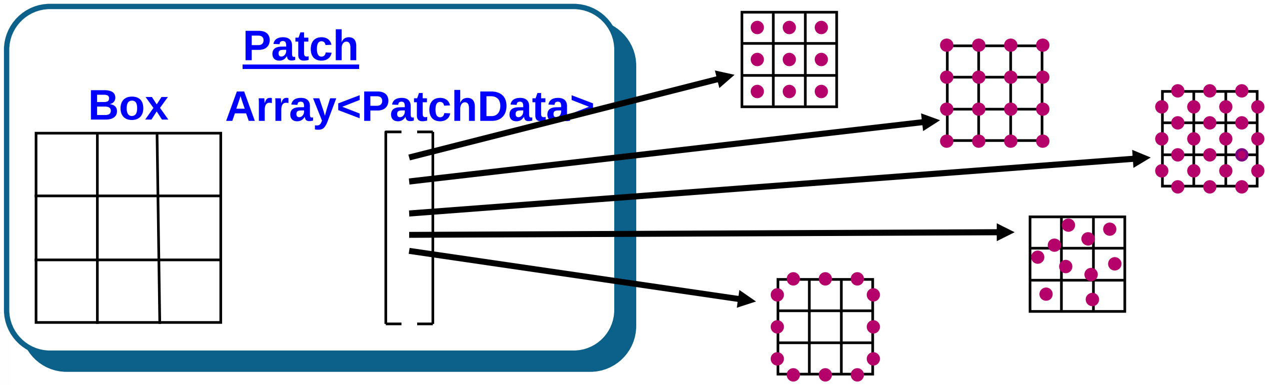
Topics covered in Part II

- SAMRAI “hierarchy” classes
  - index space
  - box
  - patch
  - level
  - patch hierarchy
- SAMRAI “variable” and “patchdata” classes
  - cell data
  - node data
  - side data
  - face data
  - edge data
  - Index data

SAMRAI “hierarchy” classes

Basic concepts to keep in mind…

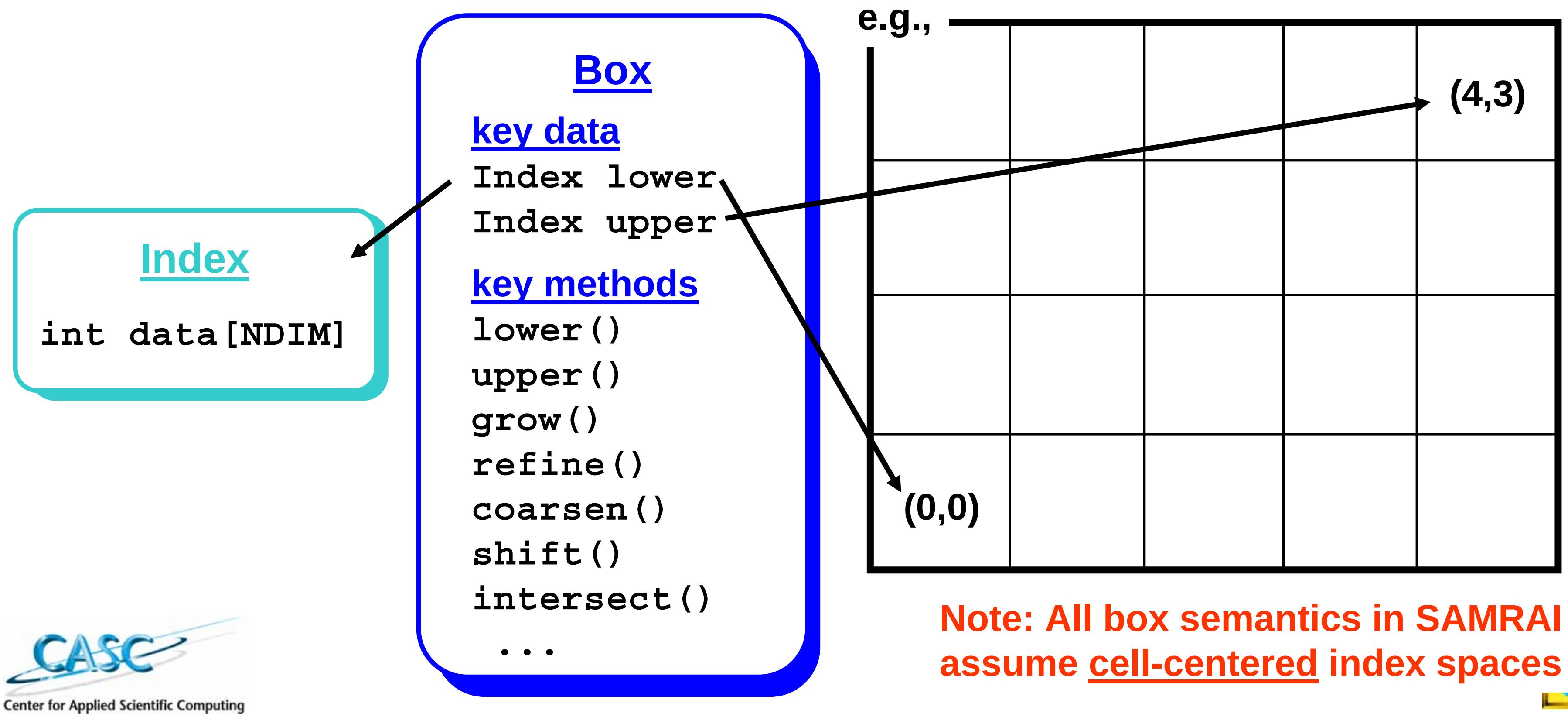
- All data operations rely on *Index* spaces and *Boxes*
- *PatchHierarchy* maintains array of *PatchLevels*
- *PatchLevel* maintains array of *Patches*
  - patches are distributed
  - index space information (boxes) is global
- *Patch* objects hold all *PatchData* objects on a *Box*



Center for Applied Scientific Computing

SAMRAI “hierarchy” classes: *Index, Box*

- **Indices, boxes, box collections (box list, box array)**
  - — all data manipulation (comp & comm) relies on index information
  - — available for 1, 2, or 3 spatial dimensions



Index

```
int data[NDIM]
```

Box

key data

```
Index lower
Index upper
```

key methods

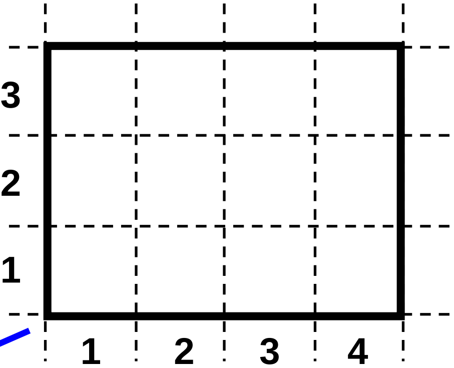
```
lower()
upper()
grow()
refine()
coarsen()
shift()
intersect()
  ...
```

Note: All box semantics in SAMRAI assume cell-centered index spaces

Any *Box* object may be coarsened or refined in index space

e.g.,

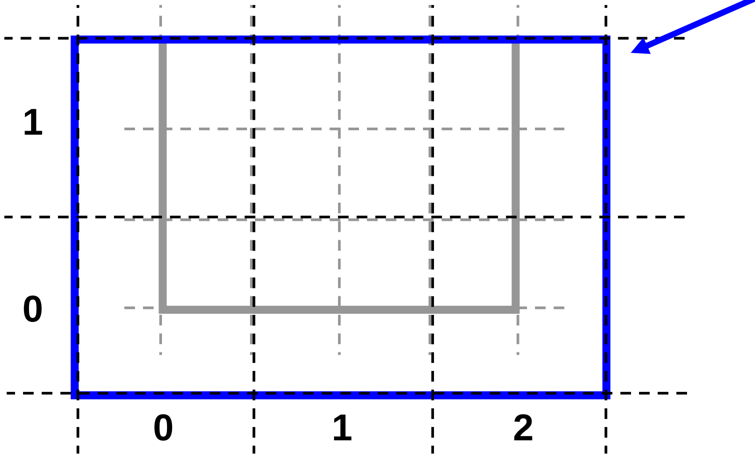
```
Box box(Index(1,1), Index(4,3))
```



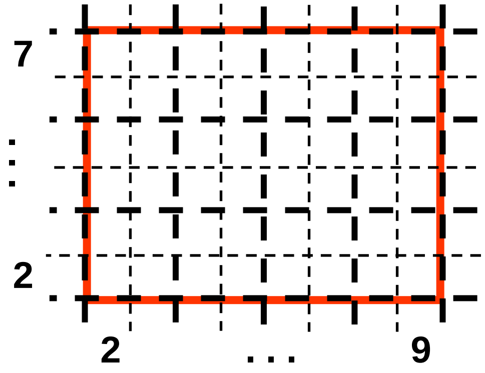
Note:

- **Coarsened box may cover larger region.**
- **Refined box will always cover same region.**

```
Box::coarsen(box, IntVector(2))
```



```
Box::refine(IntVector(2))
```



coarse lower = (lower+1)/ratio-1, lower < 0
= lower/ratio , otherwise
coarse upper = (upper+1)/ratio-1, upper < 0
= upper/ratio , otherwise

fine lower = lower * ratio
fine upper = upper * ratio + ratio - 1

SAMRAI “hierarchy” classes: *PatchHierarchy, PatchLevel, Patch*

- “PatchHierarchy” holds an array of “PatchLevel”s
  - “PatchLevel” holds an array of “Patch”es
    - “Patch” holds an array of “PatchData”

PatchHierarchy

key data

```
Array<PatchLevel>levels
```

key methods

```
getNumberLevels()
getLevel()
makeNewPatchLevel()
removePatchLevel()
  ...
```

PatchLevel

key data

```
BoxArray phys_domain
BoxArray boxes
Array<Patch> patches
```

key methods

```
getPhysicalDomain()
getBoxes()
getRatio()
getPatch()
allocatePatchData()
dealloatePatchData()
 ...
```

Patch

key data

```
Box box
Array<PatchData> data
```

key methods

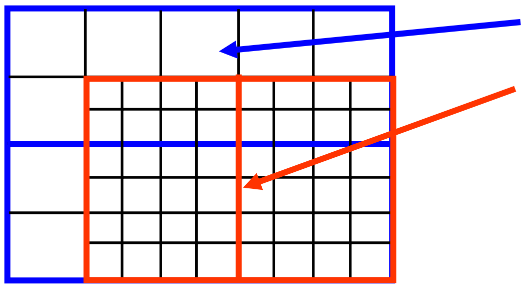
```
getBox()
getPatchData()
allocatePatchData()
deallocatePatchData()
  ...
```

Center for Applied Scientific Computing

Each *PatchLevel* object owns “local” *Patch*es and all “global” *Box* information

e.g.,

`PatchHierarchy hierarchy`

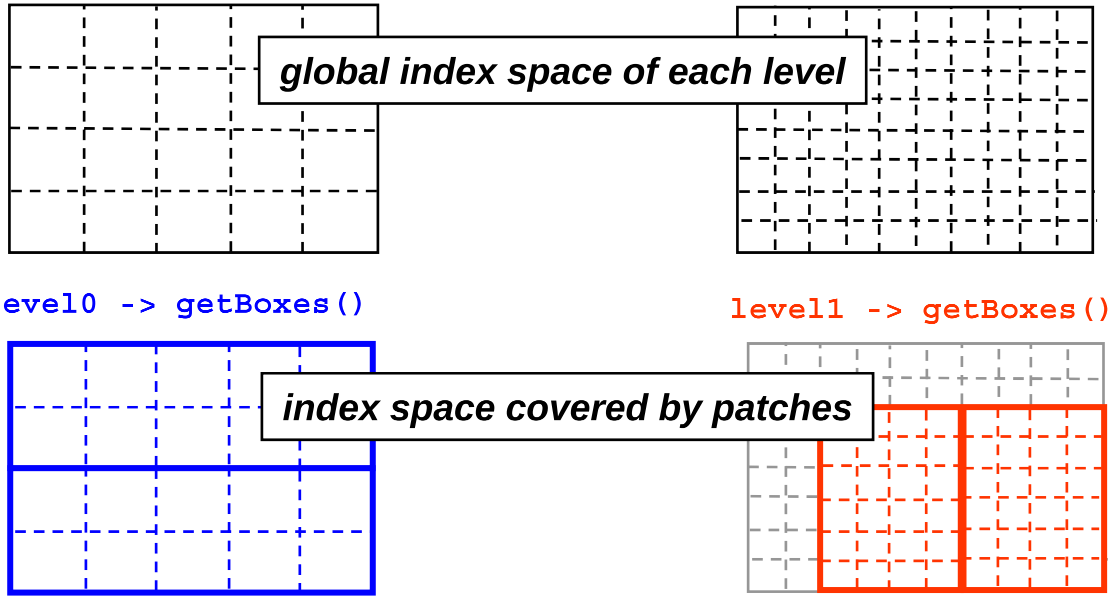


```
Pointer<PatchLevel> level0 = hierarchy.getLevel(0)
Pointer<PatchLevel> level1 = hierarchy.getLevel(1)
```

Global (i.e., shared) box information

`level0 -> getPhysicalDomain()`

`level1 -> getPhysicalDomain()`



global index space of each level

`level0 -> getBoxes()`

`level1 -> getBoxes()`

index space covered by patches

Note: Patches (and thus data) are distributed across processors, but each processor knows all domain and box information

SAMRAI “variable” and “patch data” classes

Center for Applied Scientific Computing

SAMRAI *Variable* and *PatchData* objects separate "static" and "dynamic" concepts

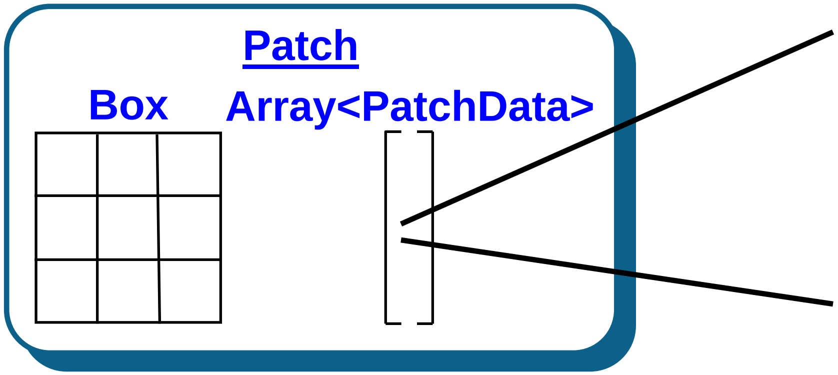
Solution algorithms and variables tend to be <u>static</u>

- **Variable**
  - defines a data quantity; type, (centering), (depth)
  - abstract base class (interface) attributes:
    - name (string)
    - unique instance id (int)
    - way to create data storage
  - creates data object instances (*abstract factory*)
  - *Variable* objects usually persist throughout computation

Mesh and data objects tend to be <u>dynamic</u>

- **PatchData**
  - represents data on a "box"
  - abstract base class (interface) attributes:
    - interior box (Box)
    - exterior box (Box)
    - ghost cell width (IntVector)
  - interface for all data communication (*strategy*)
  - (usually) created by factory associated with variable
  - *PatchData* objects are created and destroyed as mesh changes

A SAMRAI *Patch* contains all data living in a box region in a level in the mesh



Patch

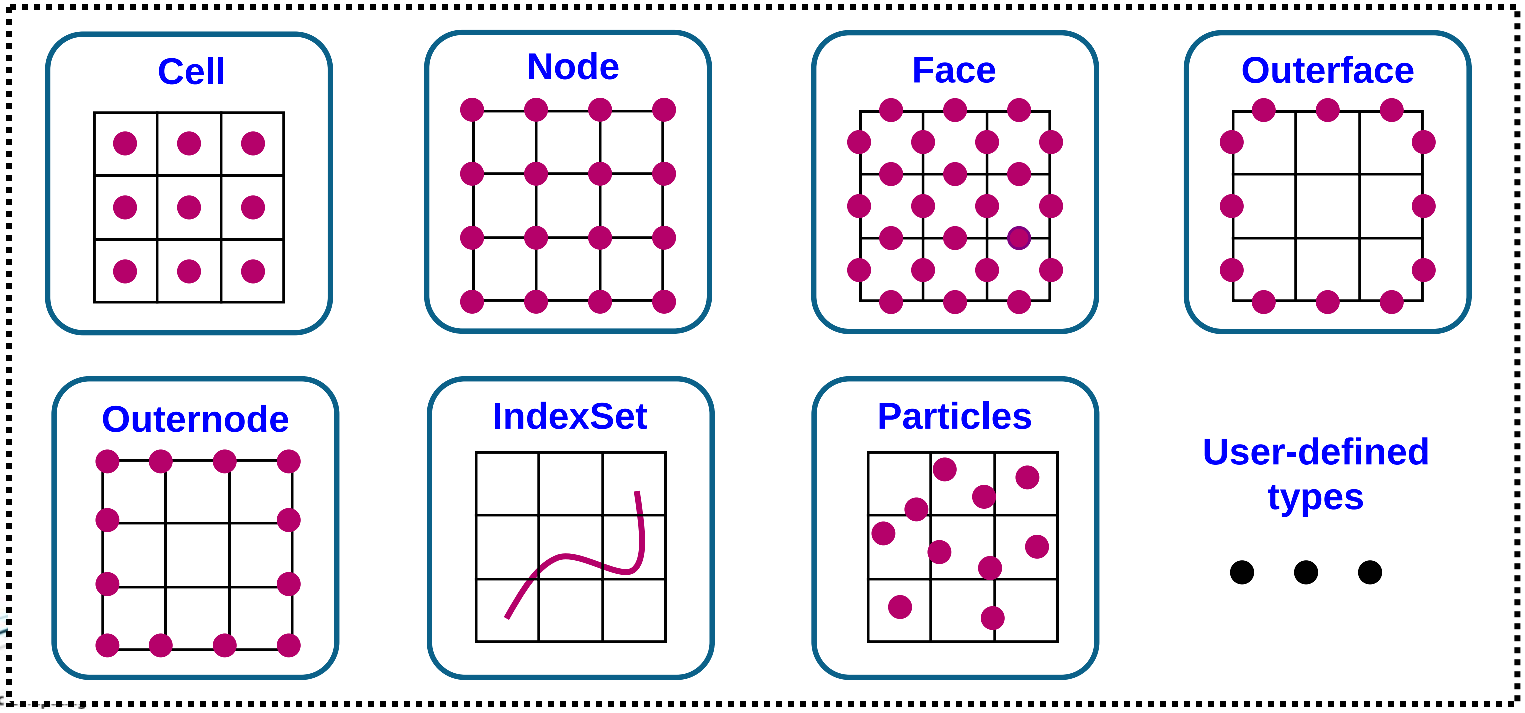
Box

Array<PatchData>

PatchData

```
getBox()
getGhostBox()
getGhostCellWidth()
copy(...)
packStream(...)
unpackStream(...)
 . . .
```

All *Patchdata* objects obey the same interface



Cell

Node

Face

Outerface

Outernode

IndexSet

Particles

User-defined types

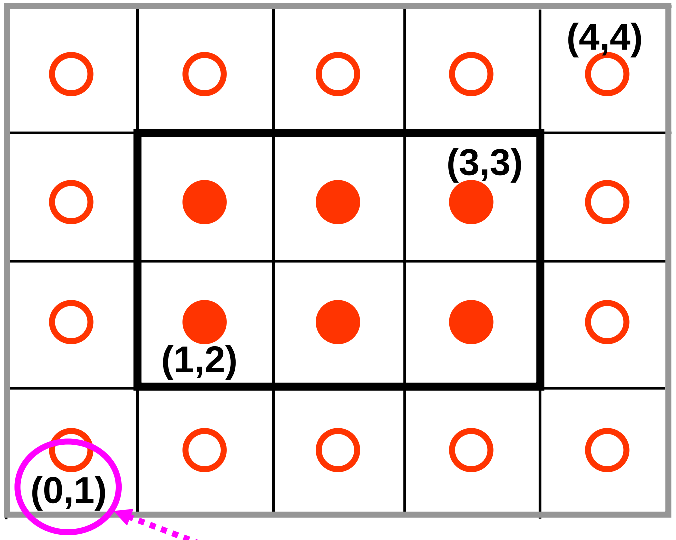
● ● ●

SAMRAI “patch data”: cell data

- *CellVariable* and *CellData* provide “cell-centered” arrays (int, float, double, dcomplex, bool, char)

2D ex.

```
CellData<double> cdat( patch.getBox(),
                       depth = 1, ghosts = (1,1) )
```



```
double* arr = cdat.getPointer()
```

Cell data array
(5 X 4 X d)

[0 : 4 ,
1 : 4 , d]

3D cell data array
Box(lower, upper)

[lower0 : upper0 ,
lower1 : upper1 ,
lower2 : upper2 , d]

`patch.getBox()` → **(1,2) - (3,3)**

`cdat.getBox()` → **(1,2) - (3,3)**

`cdat.getGhostBox()` → **(0,1) - (4,4)**

Note: data is allocated over “ghost box”

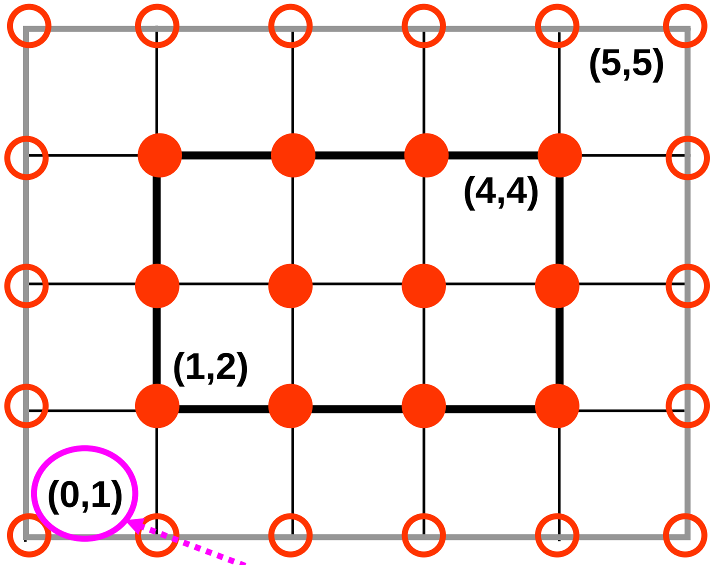
column-major (FORTRAN) ordering
lower array indices same as “ghost box”

SAMRAI “patch data”: node data

- ***NodeVariable*** **and** ***NodeData*** **provide “node-centered” arrays (int, float, double, dcomplex, bool, char)**

<u>2D ex.</u>

```
NodeData<double> ndat( patch.getBox(),
                       depth = 1, ghosts = (1,1) )
```



```
double* arr = ndat.getPointer()
```

Node data array (6 X 5 X d)

[0 : 5 ,
 1 : 5 , d]

3D node data array Box(lower, upper)

[lower0 : upper0 + 1 ,
 lower1 : upper1 + 1 ,
 lower2 : upper2 + 1 , d]

Note: index scheme for node data adds 1 to upper box index in each dimension

`patch.getBox()` → **(1,2) - (3,3)**

`ndat.getBox()` → **(1,2) - (3,3)**

`ndat.getGhostBox()` → **(0,1) - (4,4)**

Note: member functions return “cell-centered” boxes

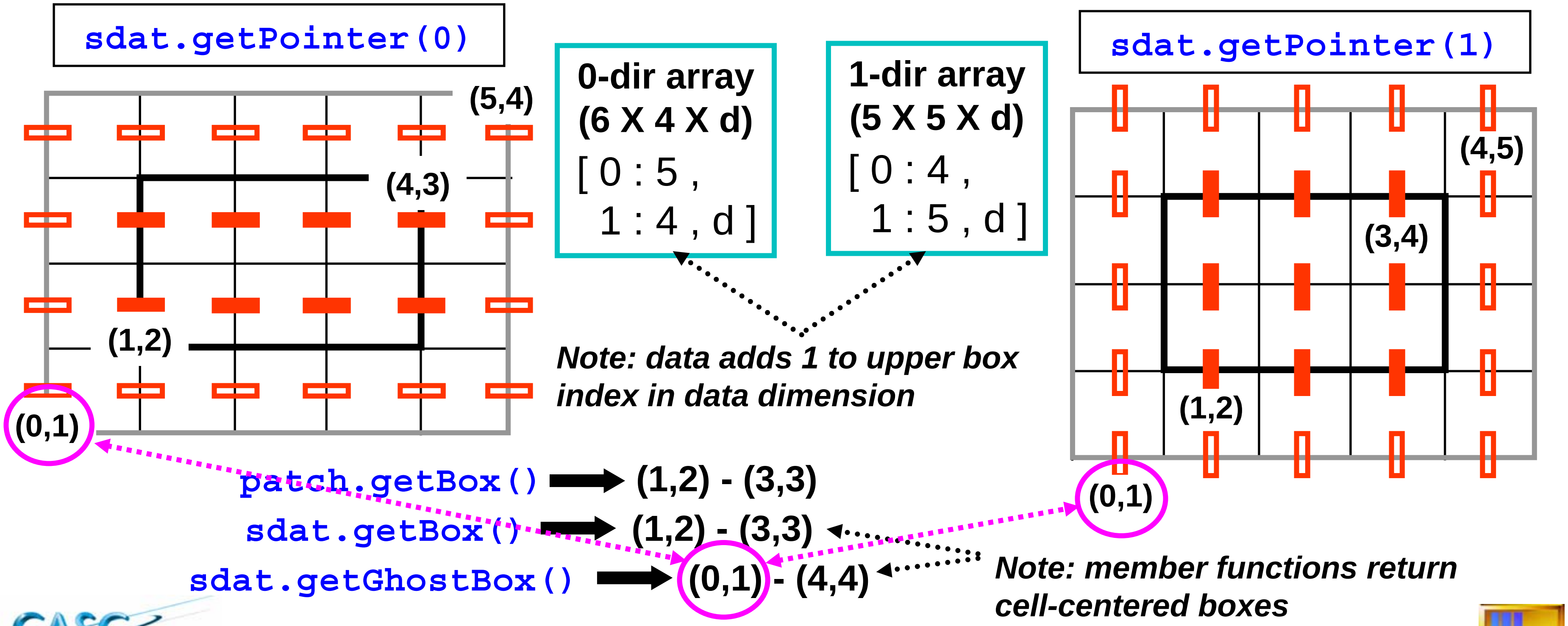
column-major (FORTRAN) ordering
lower array indices same as “ghost box”

SAMRAI “patch data”: side data

- ***SideVariable*** **and** ***SideData*** **provide “side-centered” arrays (int, float, double, dcomplex, bool, char)**

2D ex.

```
SideData<double> sdat( patch.getBox(),
                       depth = 1, ghosts = (1,1) )
```



0-dir array (6 X 4 X d)

[0 : 5 , 1 : 4 , d]

1-dir array (5 X 5 X d)

[0 : 4 , 1 : 5 , d]

Note: data adds 1 to upper box index in data dimension

`patch.getBox()` → **(1,2) - (3,3)**

`sdat.getBox()` → **(1,2) - (3,3)**

`sdat.getGhostBox()` → **(0,1) - (4,4)**

Note: member functions return cell-centered boxes

column-major (FORTRAN) ordering
lower array indices same as “ghost box”

Center for Applied Scientific Computing

SAMRAI “patch data”: side & face data

- ***FaceVariable*** **and** ***FaceData*** **arrays are similar to side**

3D <u>side data</u> arrays **Box(lower, upper)**		**3D <u>face data</u> arrays** **Box(lower, upper)**
[lower0 : upper0 + 1 , lower1 : upper1 , lower2 : upper2, d]	**0-direction (or “x”)**	[lower0 : upper0 + 1 , lower1 : upper1 , lower2 : upper2, d]
[lower0 : upper0 , lower1 : upper1 + 1, lower2 : upper2 , d]	**1-direction (or “y”)**	[lower1 : upper1 + 1 , lower2 : upper2 , lower0 : upper0 , d]
[lower0 : upper0 , lower1 : upper1 , lower2 : upper2 + 1 , d]	**2-direction (or “z”)**	[lower2 : upper2 + 1 , lower0 : upper0 , lower1 : upper1 , d]

Note: *FaceData* permutes (0,1,2); leading array dimension is spatial dimension

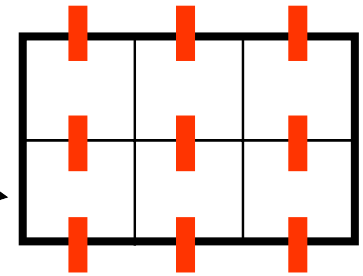
SAMRAI “patch data”: other face/side data

- *SideData* can be managed in single directions

For example

```
SideData<double> sdat( patch.getBox(),
                       depth = 1, ghosts = (0,0)
                       direction = 1 )
```

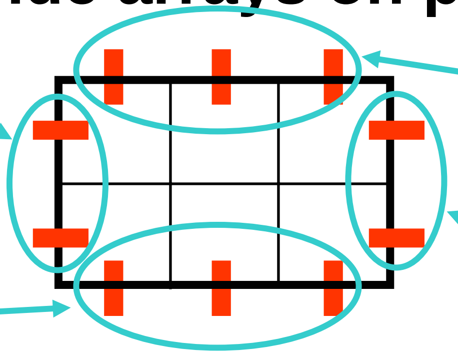
gives



`sdat.getPointer(1)`

`sdat.getPointer(0)` → (NULL pointer)

- *OuterfaceVariable* / *OuterfaceData* and *OutersideVariable* / *OutersideData* provide arrays on patch boundaries



`osdat.getPointer(0,0)`

`osdat.getPointer(1,0)`

direction

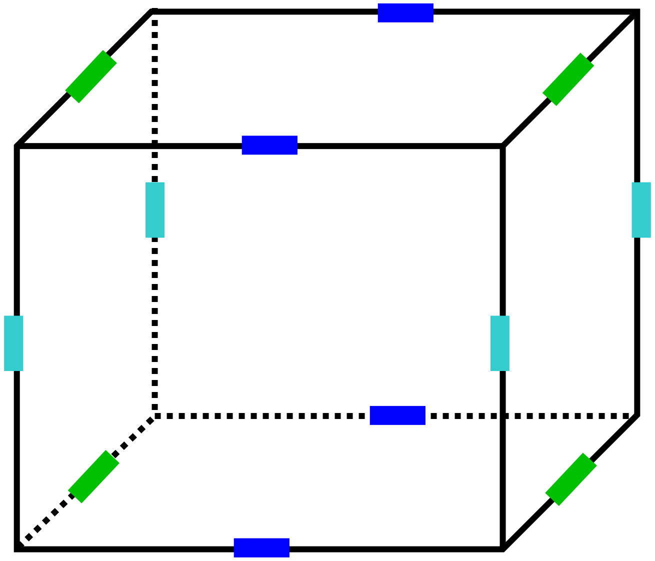
`osdat.getPointer(1, 1)`

`osdat.getPointer(0, 1)`

lower/upper

SAMRAI "patch data": edge array data

- ***EdgeVariable* and *EdgeData* provide edge-centered arrays**
  - like side & face data, edge data uses NDIM arrays; axis corresponds to edges parallel to axis direction (recall side/face axis corresponds to side/face with normal in axis direction)
  - in 3D :



3D edge data arrays
Box(lower, upper)

[lower0 : upper0 ,
lower1 : upper1 + 1 ,
lower2 : upper2 + 1 , d]

[lower0 : upper0 + 1 ,
lower1 : upper1 ,
lower2 : upper2 + 1 , d]

[lower0 : upper0 + 1,
lower1 : upper1 + 1,
lower2 : upper2 , d]

SAMRAI "patch data": index data

- *IndexVariable* and *IndexData* are template classes to manage quantities on irregular cell-centered index sets

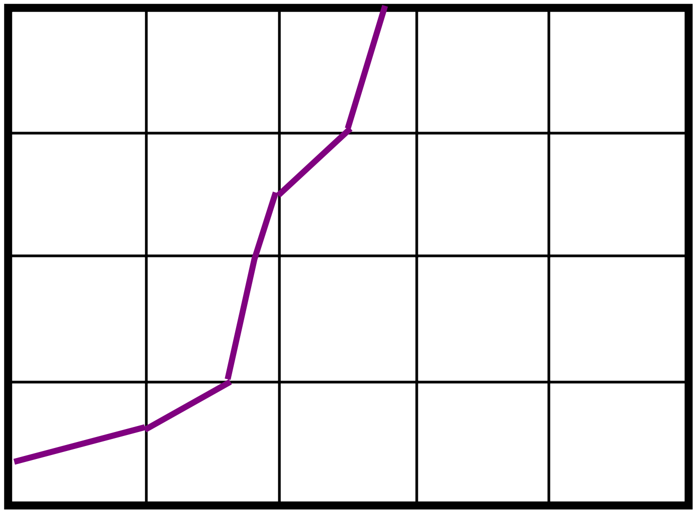
```
IndexVariable<TYPE> ivar("name")
IndexData<TYPE> idata(Box& box, ghosts)
```

"TYPE"

Required methods

```
TYPE()
TYPE& operator=(const TYPE&)
getDataStreamSize(Box&)
packStream(...)
unpackStream(...)
```

e.g.



CutCell type describes internal boundary and state information along boundary

SAMRAI supports new patch data types via inheritance without recompilation

- Create a MyData subclass and provide virtual functions

```
class MyData : public PatchData
{
      void copy(...);
      void packStream(...);
      int getDataStreamSize(...)
      . . .
};
```

- Create a MyFactory subclass to allocate MyData objects

```
class MyFactory : public PatchDataFactory
{
      Pointer<PatchData> allocate(...);
      . . .
};
```

- Create MyVariable subclass to create MyFactory objects

```
class MyVariable : public Variable
{
      Pointer<PatchDataFactory> getPatchDataFactory(...);
      . . .
};
```

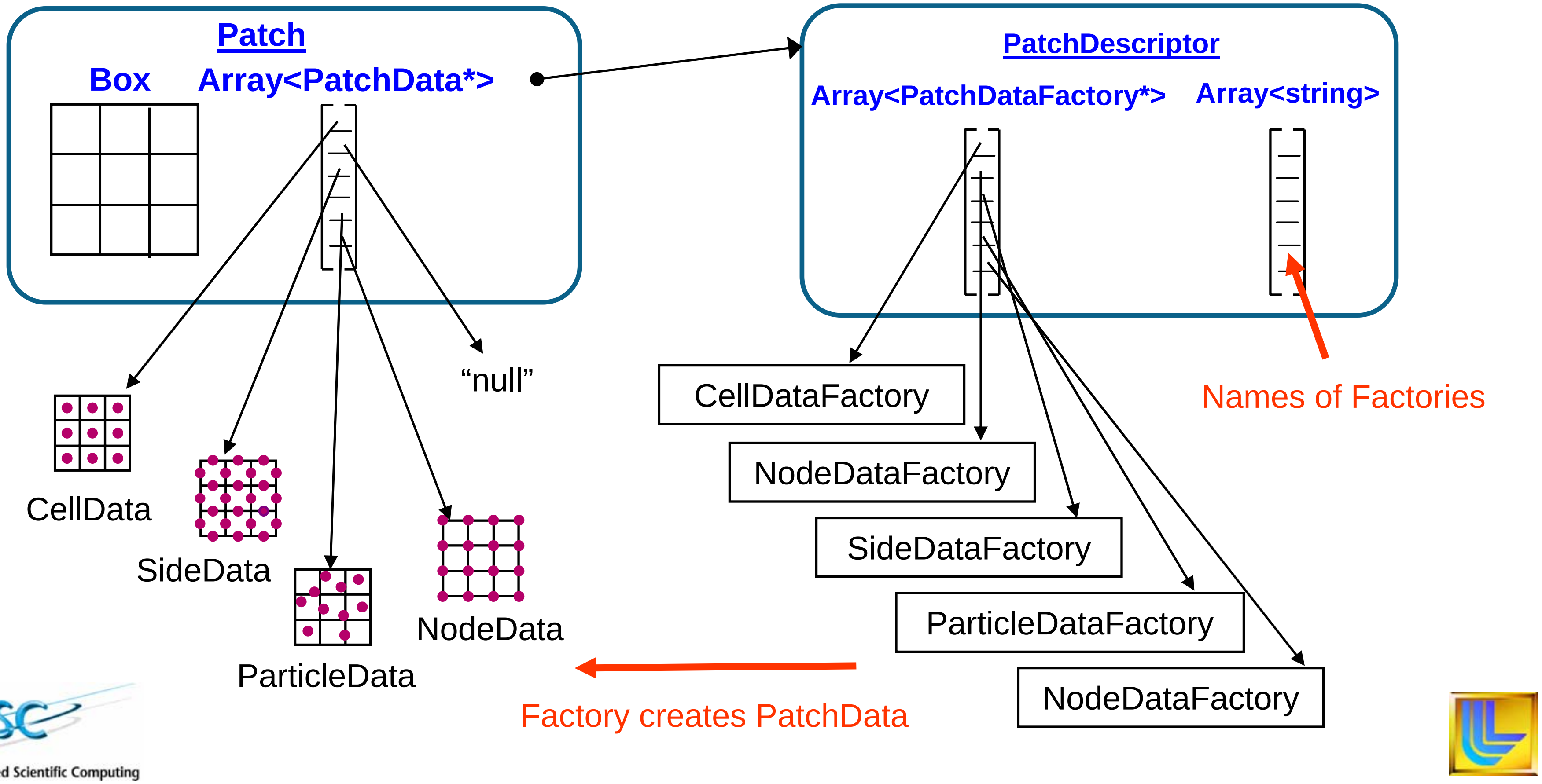
Part III

Topics covered in Part III

- SAMRAI “Variable Database”
  - motivation
  - usage

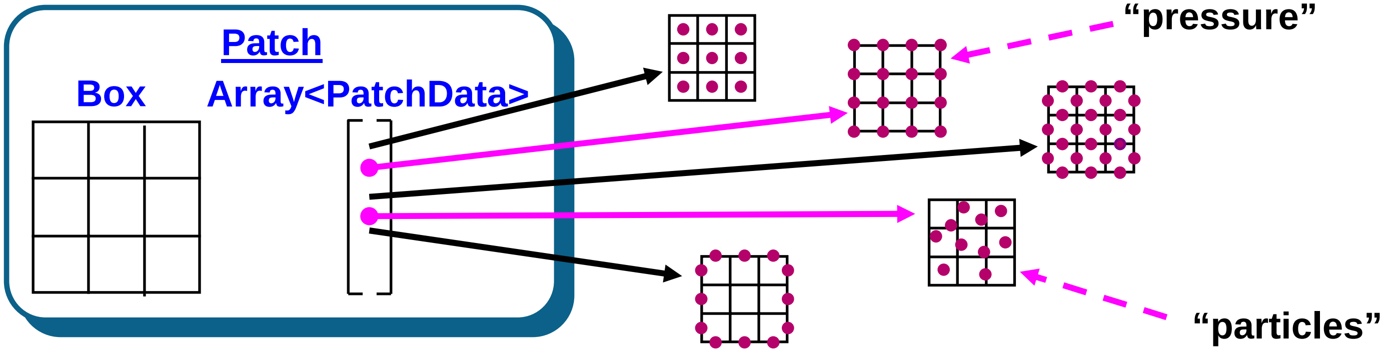
Important note: VariableDatabase is not needed for nearly all SAMRAI functionality

Basic model for SAMRAI data management: **one-to-one** correspondence between PatchData objects (owned by patches) and PatchDataFactory objects (owned by PatchDescriptor). **One PatchDescriptor instance shared by all patches.**



The *VariableDatabase* helps to manage variables and data on patch hierarchy

- **Recall: a *Patch* contains all data on a *Box* region**



  - *Variables* define mesh quantities; used to create *PatchData*
  - each patch data item lives at the same patch data array index on every patch

- ***VariableDatabase* holds variable-patch data mappings**
  - *Singleton* object; one instance, accessible everywhere in code
  - provides variable "contexts" →multiple storage locations/variable
  - provides access to shared patch descriptor object (consistency)

Important: database contents are defined and set by user

“Variable database” motivation

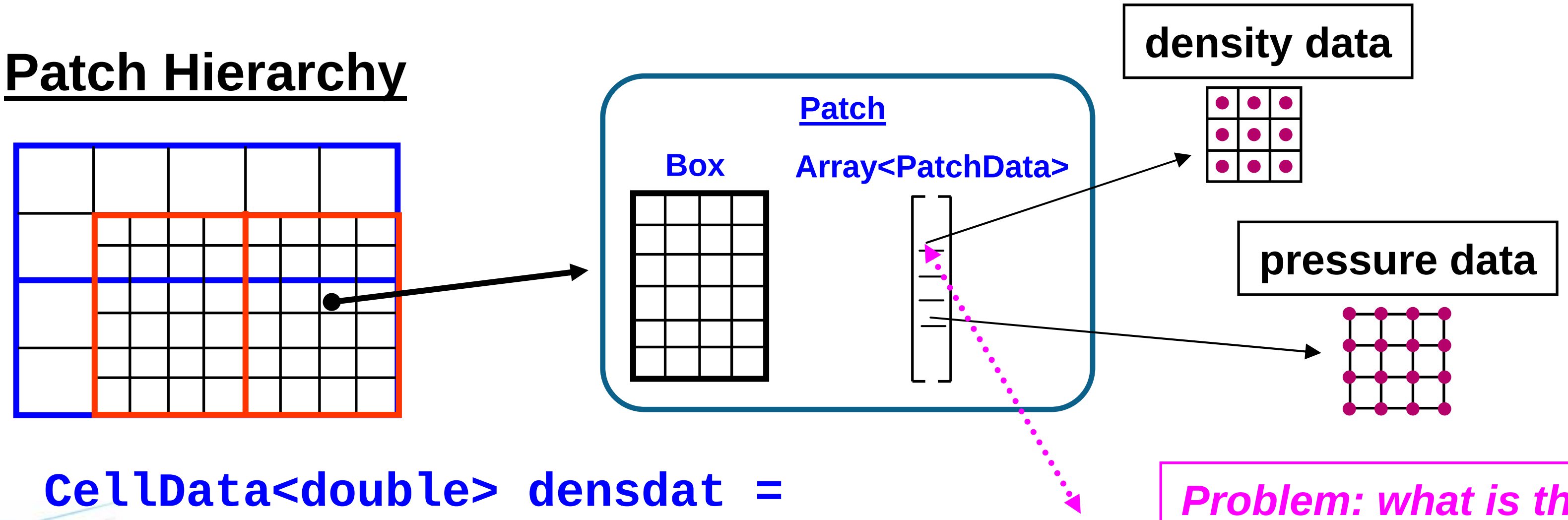
VariableDatabase motivation I: setting data slots for variables on patches

- After creating a variable, we need to establish storage slot(s) for variable data on each patch

Application code

```
CellVariable<double> density(“density”, depth = 1)
NodeVariable<double> pressure(“pressure”, depth = 1)
 . . .
```

Patch Hierarchy



```
CellData<double> densdat =
                 patch.getPatchData(??)
```

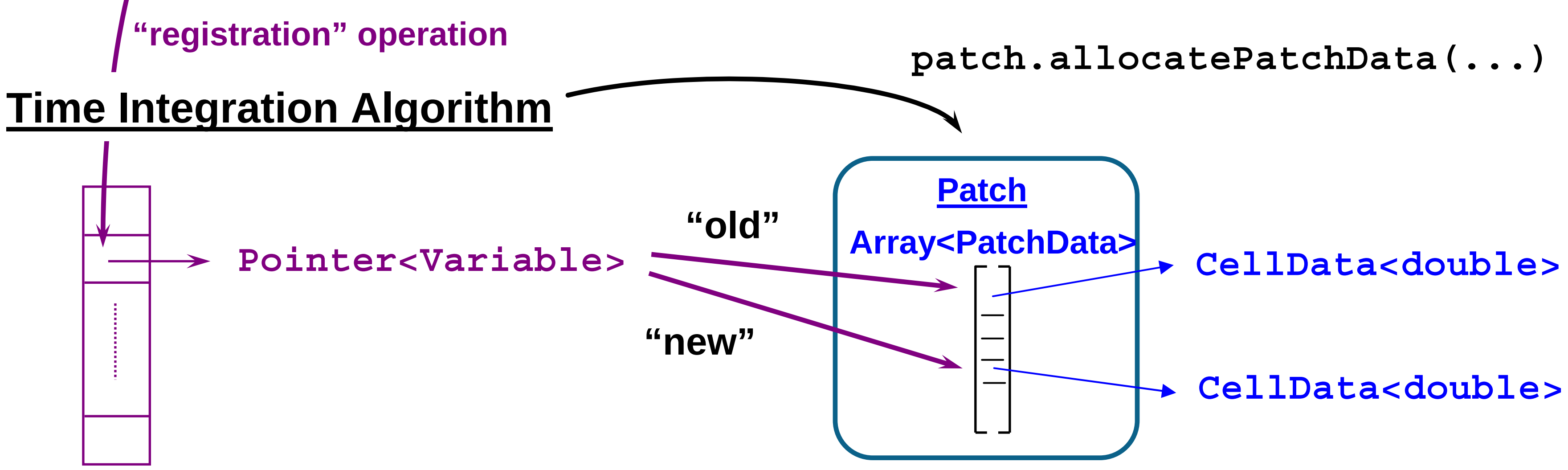
Problem: what is the data array index?

VariableDatabase motivation II: using multiple data entries for one variable

- Integration algorithms may require multiple data “contexts” for each variable

Application code

```
CellVariable<double> density(“density”, depth = 1)
 . . .
```



“registration” operation

`patch.allocatePatchData(...)`

Time Integration Algorithm

`Pointer<Variable>`

“old”

“new”

Patch

Array<PatchData>

`CellData<double>`

`CellData<double>`

Problem: how do we implement a general algorithm that manages an arbitrary set of time dependent variables with “old” and “new” storage?

VariableDatabase motivation III: sharing variables between different algorithms

- A variable may be used differently in different parts of the solution procedure

e.g., Application code

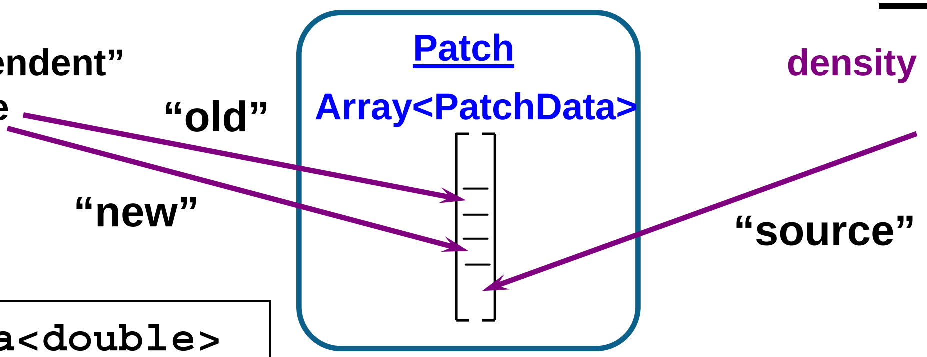
```
CellVariable<double> density("density", depth = 1)
 . . .
```

Solver A

density is a "time-dependent" solution variable

Solver B

density is a "source term" variable



Note: Each `CellData<double>` *object may have different storage needs (e.g., ghost cell width)*

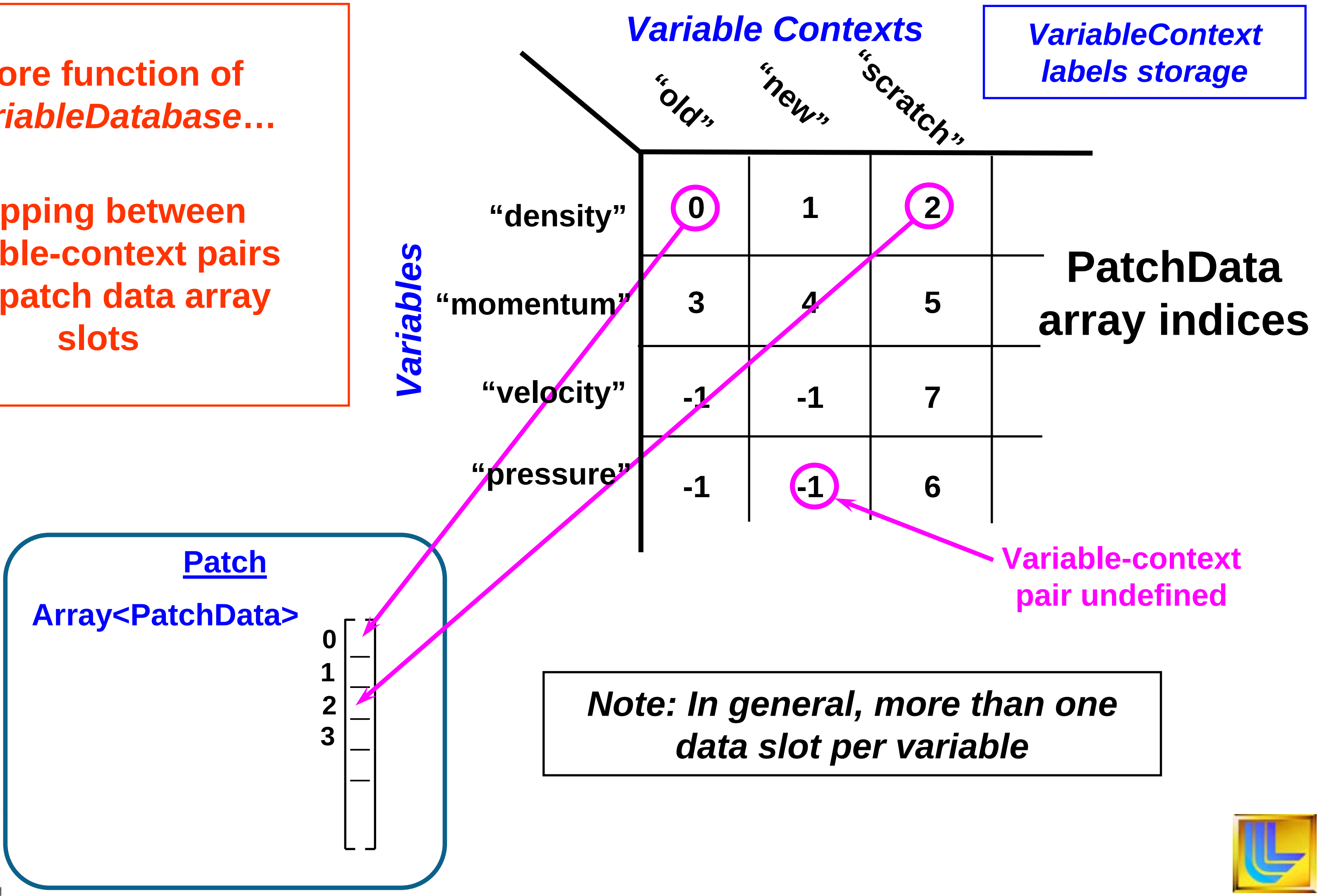
Problem: how can solvers share variables and data <u>and</u> manage data independently?

“Variable database” usage

Conceptual view of *VariableDatabase* and *VariableContext*

Core function of *VariableDatabase*…

Mapping between variable-context pairs and patch data array slots



Variable Contexts

VariableContext labels storage

Variables

	"old"	"new"	"scratch"
"density"	0	1	2
"momentum"	3	4	5
"velocity"	-1	-1	7
"pressure"	-1	-1	6

PatchData array indices

Variable-context pair undefined

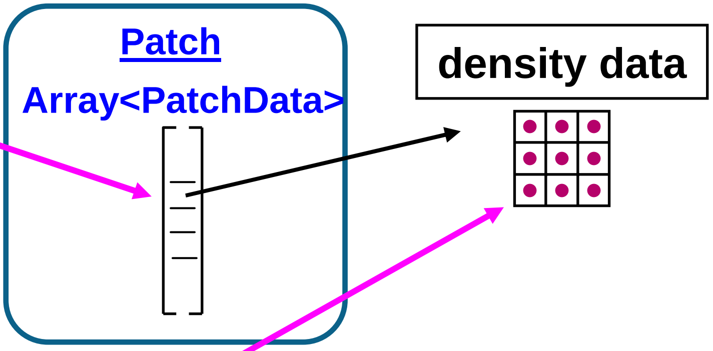
Note: In general, more than one data slot per variable

VariableDatabase usage I: setting storage slot for patch data (no *VariableContext*)

```
// get pointer to cell-centered density variable
Pointer< CellVariable<double> > density = . . .

// get pointer to Singleton VariableDatabase object
VariableDatabase* vdb = VariableDatabase::getDatabase()

// get array index  for density data (default factory)
int dens_id = vdb->registerPatchDataIndex(density)
```



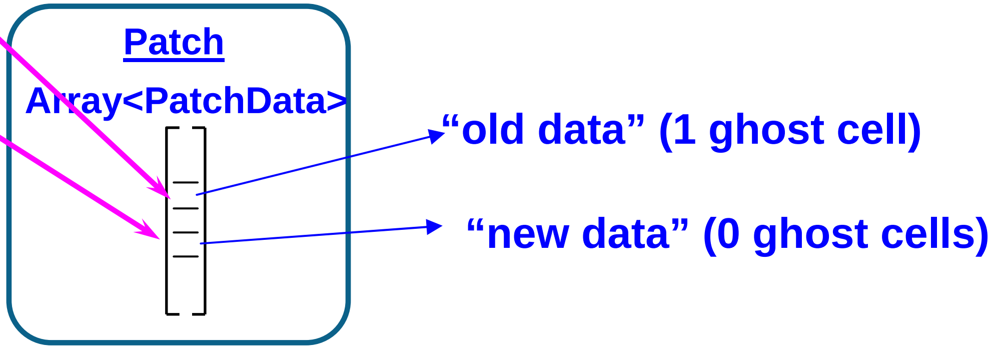
```
// get density data on patch
Pointer< CellData<double> > dens_data = patch.getPatchData(dens_id)
```

VariableDatabase usage II: multiple storage slots via *VariableContexts*

```
// pointer to some variable and Singleton VariableDatabase
Pointer<Variable> var = . . .
VariableDatabase* vdb = VariableDatabase::getDatabase()

// get pointers to “OLD” and “NEW” VariableContext objects
Pointer<VariableContext> old_ctxt = vdb->getContext(“OLD”)
Pointer<VariableContext> new_ctxt = vdb_>getContext(“NEW”)

// set “OLD” and “NEW” patch data locations
int old_var_id = vdb->registerVariableAndContext(var, old_ctxt,
                                                 IntVector(1))
int new_var_id = vdb->registerVariableAndContext(var, new_ctxt,
                                                 IntVector(0))
```



Patch

Array<PatchData>

“old data” (1 ghost cell)

“new data” (0 ghost cells)

VariableDatabase usage III: sharing variables and data in different algorithms

```
CellVariable<double> density(“density”, depth = 1)
VariableDatabase* vdb = ...
vdb->addVariable(density)
```

VariableDatabase allows global access to *Variable*

Solver A

(density is “time-dependent”)

```
VariableDatabase* vdb = ...
int dens_old_id = vdb->registerVariableAndContex( vdb->getVariable(“density”),
                                                  vdb->getContext(“OLD”),
                                                  IntVector(2) )
int dens_new_id = vdb->registerVariableAndContex( vdb->getVariable(“density”),
                                                  vdb->getContext(“NEW”),
                                                  IntVector(0) )
```

Solver B

(ndensity is a “source term”)

```
VariableDatabase* vdb = ...
int dens_src_id = vdb->registerVariableAndContex( vdb->getVariable(“density”),
                                                  vdb->getContext(“SOURCE”),
                                                  IntVector(1) )
```

Summary of *VariableDatabase* usage

registration also adds variable

- **Using variables and variable contexts**
  1 Add variable: `addVariable(Pointer<Variable>)`
  2 Get variable context: `getContext(string&)`
  3 Define variable-context pair mapping to data index:

```
int registerVariableAndContext(Pointer<Variable>,
                               Pointer<VariableContext>,
                               IntVector& ghost_width)
```

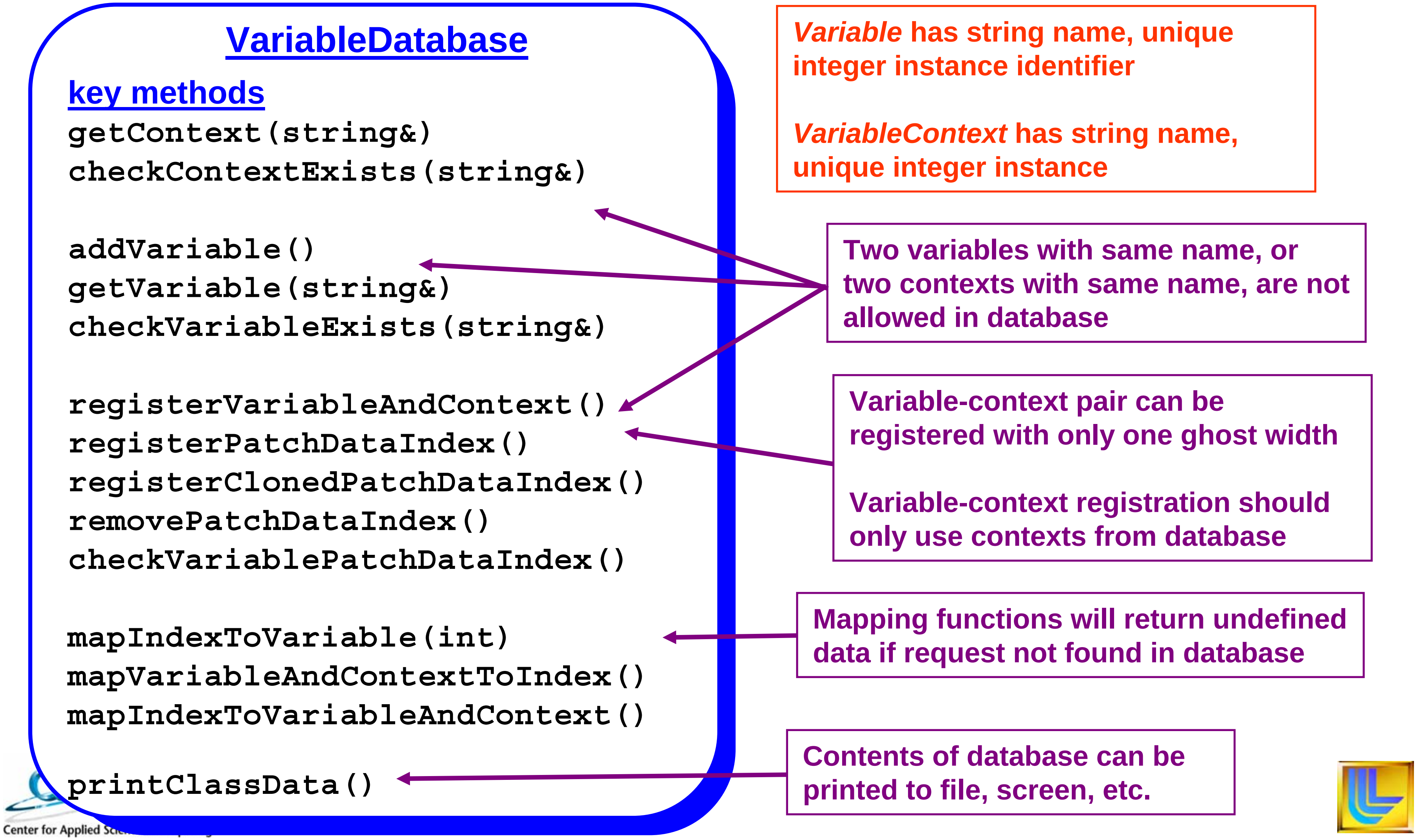
  4 Map between variable-context pairs and data indices:
  `mapVariableAndContextToIndex(), mapIndexToVariableAndContext()`

- **Using variables and data indices only (no contexts)**
  1 add variable: `addVariable(Pointer<Variable>)`
  2 Define/undefine variable mapping to data index:

```
int registerPatchDataIndex(Pointer<Variable>, int data_id = -1)
int registerClonedPatchDataIndex(Pointer<Variable>, int old_id)
void removePatchDataIndex(int data_id)
```

Map data index to variable: `mapIndexToVariable(int)`

VariableDatabase helps to maintain consistent variable-data management



VariableDatabase

key methods

```
getContext(string&)
checkContextExists(string&)

addVariable()
getVariable(string&)
checkVariableExists(string&)

registerVariableAndContext()
registerPatchDataIndex()
registerClonedPatchDataIndex()
removePatchDataIndex()
checkVariablePatchDataIndex()

mapIndexToVariable(int)
mapVariableAndContextToIndex()
mapIndexToVariableAndContext()

printClassData()
```

Variable has string name, unique integer instance identifier

VariableContext has string name, unique integer instance

Two variables with same name, or two contexts with same name, are not allowed in database

Variable-context pair can be registered with only one ghost width

Variable-context registration should only use contexts from database

Mapping functions will return undefined data if request not found in database

Contents of database can be printed to file, screen, etc.

Part IV

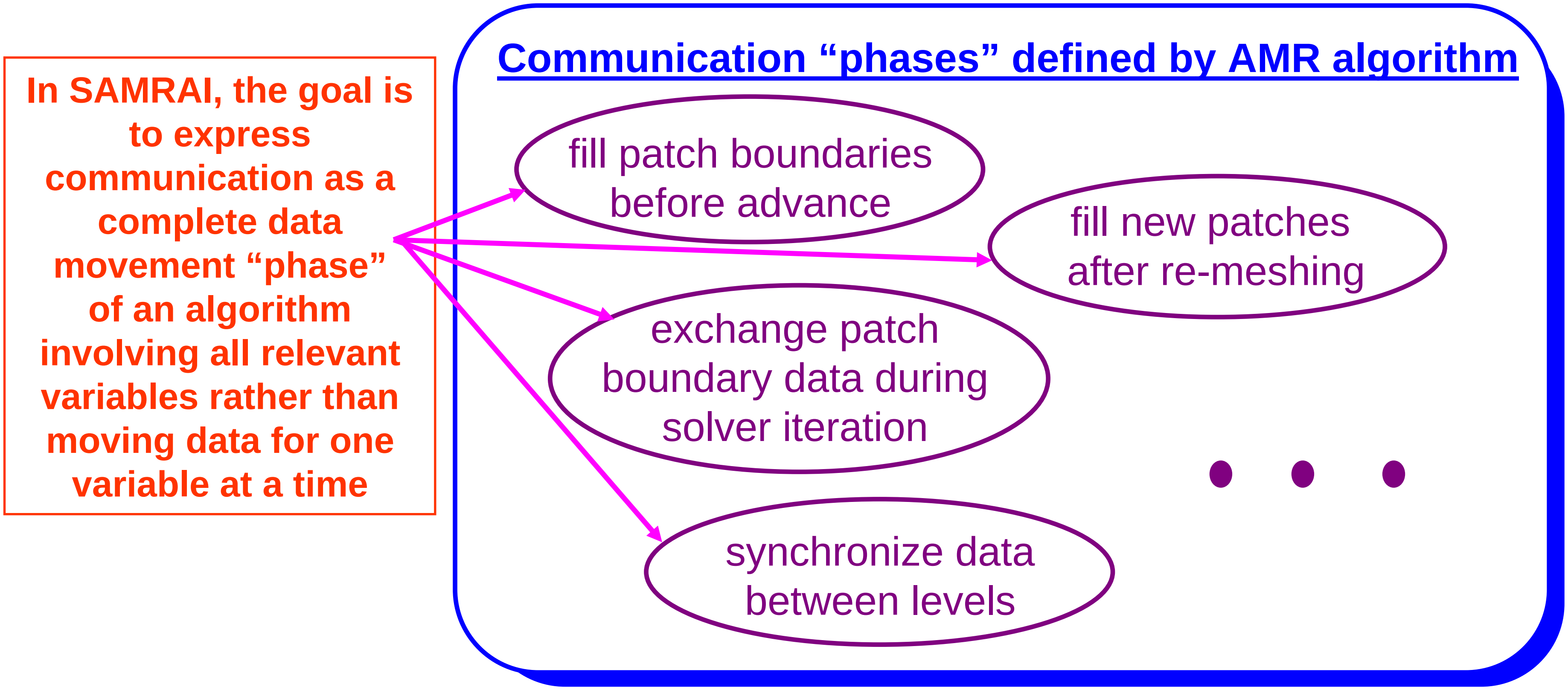
Topics covered in Part IV

- Communicating data on an AMR patch hierarchy using SAMRAI
  - SAMRAI design motivation and concepts
  - Refine Algorithm and Refine Schedule
  - Coarsen Algorithm and Coarsen Schedule

SAMRAI parallel data communication motivation and concepts

Center for Applied Scientific Computing

SAMRAI data transfer model captures general AMR communication patterns



In SAMRAI, the goal is to express communication as a complete data movement "phase" of an algorithm involving all relevant variables rather than moving data for one variable at a time

Communication "phases" defined by AMR algorithm

fill patch boundaries before advance

fill new patches after re-meshing

exchange patch boundary data during solver iteration

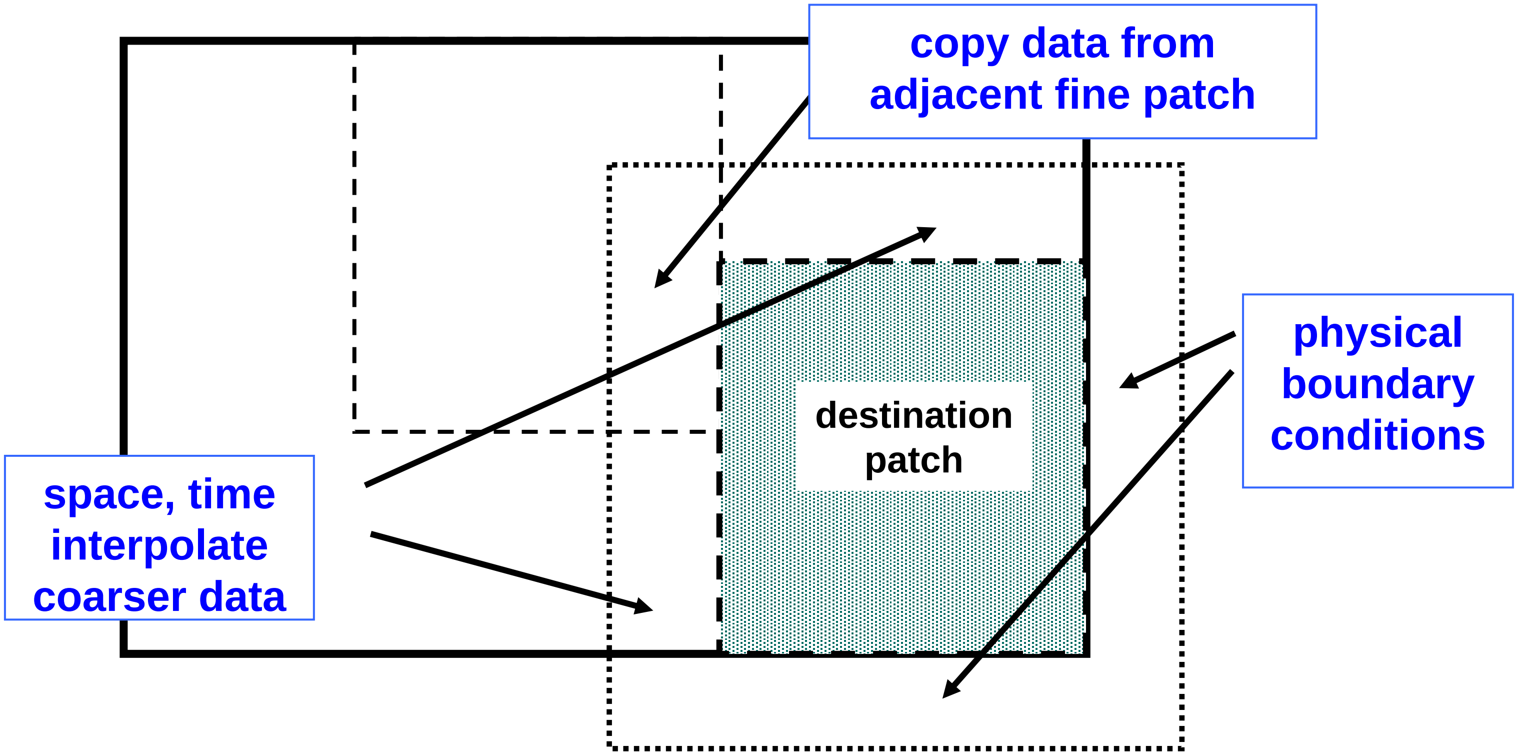
synchronize data between levels

• • •

- **Each scenario involves a set of variables and operations**
- **Operations (spatial coarsen/refine, time interpolation, boundary conditions) depend on mesh geometry, data centering, data type**

Data manipulation is dictated by solution algorithm and application needs

For example, before performing numerical operations on a patch, “ghost cell” data values are set



copy data from adjacent fine patch

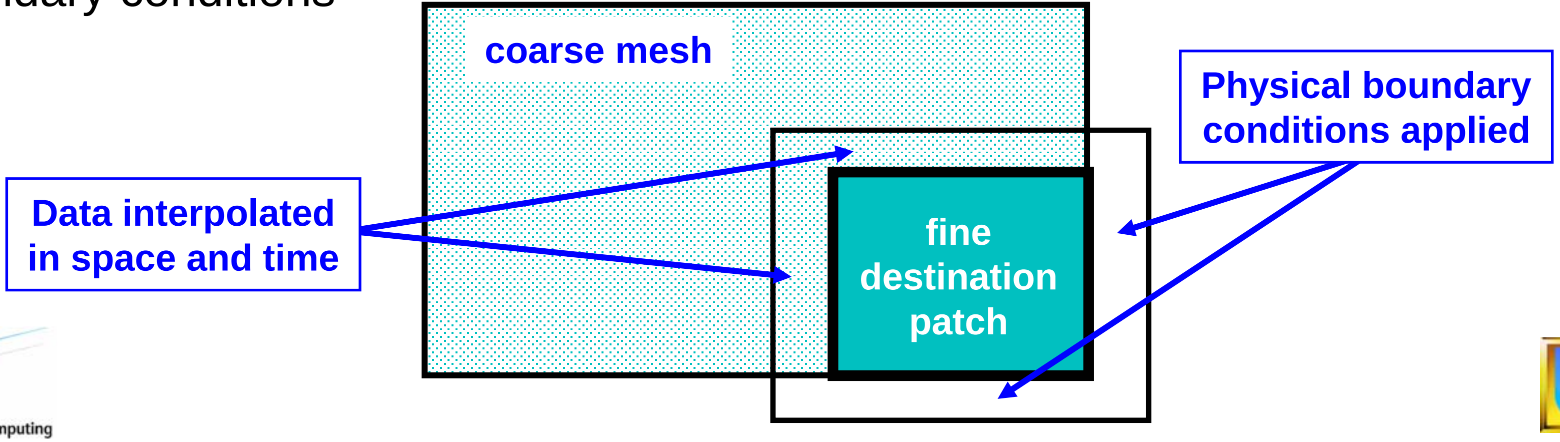
physical boundary conditions

space, time interpolate coarser data

SAMRAI framework view: SAMR data movement involves arbitrary combinations of variable quantities and operations

SAMRAI communication framework centers around three abstractions

- **Communication Algorithm** supports high-level description of data transfer phases of numerical solution algorithms
  - expressed using variables, coarsen/refine operators, etc.
  - independent of AMR mesh configuration
- **Communication Schedule** manages transactions to execute data transfers
  - automatically treats complexity of different data types (e.g., centerings)
  - depends on AMR mesh configuration
- **"Patch Strategy"** is interface to user-supplied coarsen/refine operations and boundary conditions


coarse mesh
Physical boundary conditions applied
Data interpolated in space and time
fine destination patch

Communication *Algorithm* and *Schedule* separate “static” and “dynamic” concepts

Solution algorithms and variables tend to be <u>static</u>

- Communication Algorithm
  - describes data transfer phase of computation
  - expressed using variables, operators, …
  - independent of mesh
  - typically persists throughout computation

Mesh and data objects tend to be <u>dynamic</u>

- Communication Schedule
  - manages details of data movement on mesh
  - created by communication algorithm
  - depends on mesh
  - re-created when mesh changes

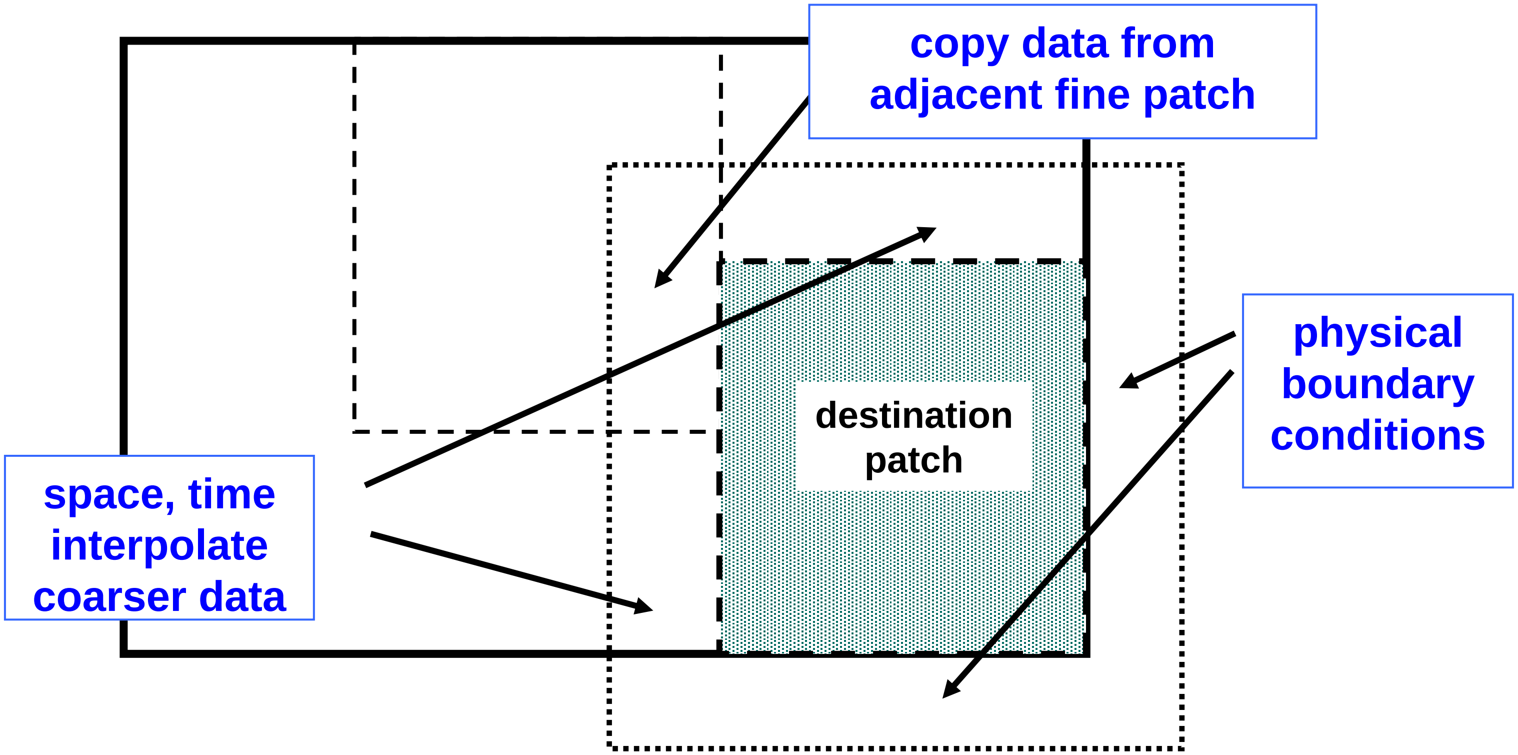
Compare with...

- *Variable*
  - defines a data quantity independent of mesh
  - usually persists throughout computation
- PatchData
  - represents data on a “box”
  - created and destroyed as mesh changes

SAMRAI “Refine Algorithm” and “Refine Schedule”

Refine Algorithm manages a data refinement phase of computation

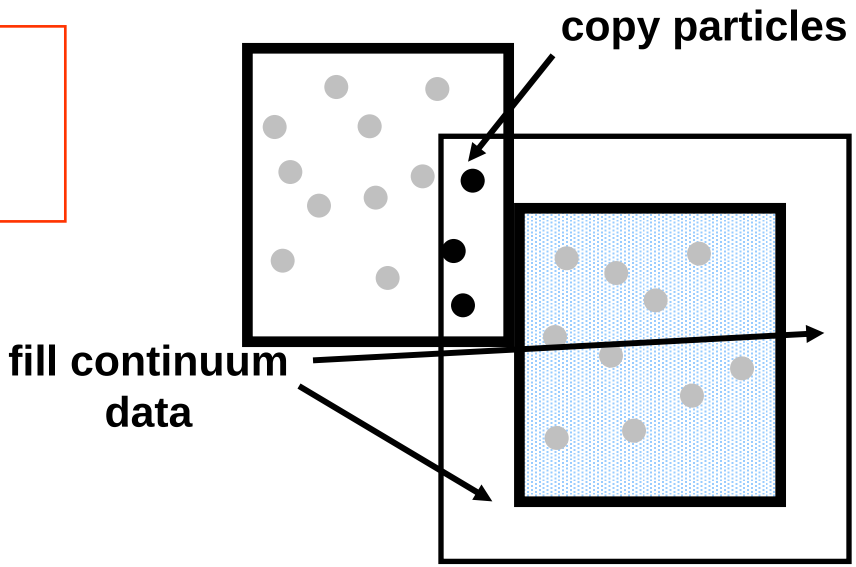
For example, before performing numerical operations on a patch, “ghost cell” data values are set


copy data from adjacent fine patch
physical boundary conditions
destination patch
space, time interpolate coarser data

SAMRAI framework supports data refinement involving arbitrary combinations of variable quantities and operations within a single data transfer.

Patch data quantities to be transferred are registered with Refine Algorithm

For example, integration of particle regions requires both continuum and particle boundary data for each patch



copy particles

fill continuum data

- Create algorithm to fill data

```
RefineAlgorithm  fill_alg;
```

- Register variable operations with algorithm:
  - density refined from coarser, copied from fine, BCs set

```
fill_alg.registerRefine(rho_old,                // destination
                        rho_old, rho_new,       // sources
                        ..., “CONSERVATIVE_INTERP”);
```

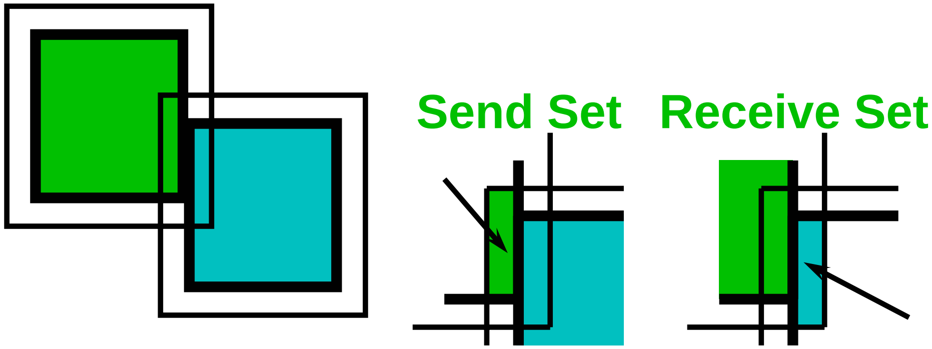
  - particles copied from neighboring patches

```
fill_alg.registerRefine(particles,   // destination
                        particles,   // source
                        ...);
```

Refine Algorithm creates Refine Schedule which computes & stores data dependencies

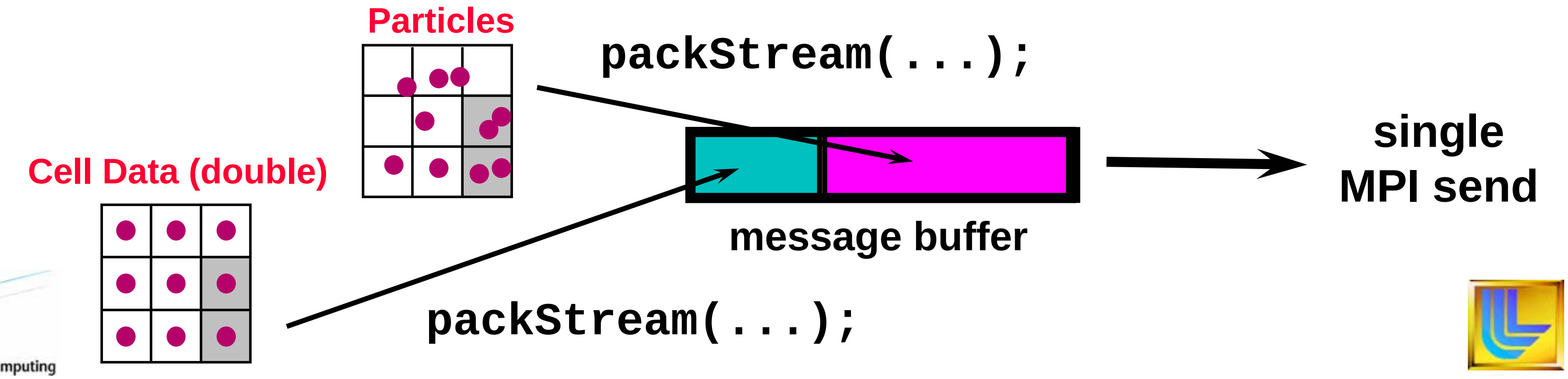
- Schedule creation constructs and stores data source and destination information needed to communicate data
  - Create schedule to fill data

```
RefineSchedule fill_sched =
  fill_alg.createSchedule(
        hierarchy, level, ...);
```



- After schedule is created, it is used to communicate data
  - Invoke data fill operations

```
fill_sched.fillData(time, ...);
```



Using *RefineAlgorithm*, *RefineSchedule* to refine data on a patch hierarchy

1 Create a ***RefineAlgorithm*** object

2 Register data transfer and refinement operations with ***RefineAlgorithm***

   – specify source and destination patch data indices

   – specify nec. spatial refinement and time interpolation operators

3 After all transfer operations are registered, create a ***RefineSchedule*** object

   – ***RefineSchedule*** depends on ***RefineAlgorithm*** object and patch hierarchy configuration

   – a ***RefinePatchStrategy*** object is needed for user-defined refinement operations and physical boundary conditions

4 Invoke the ***RefineSchedule*** to perform data refinement and transfer operations

Notes on using refine algorithms and refine schedules I

- ***RefineAlgorithm/Schedule*** objects are used to refine data in AMR hierarchy and to copy data between patches on two levels (may or may not be part of same hierarchy). Note that we consider copy operations to be a special case of refine operations.
- ***RefineAlgorithm*** has two **`registerRefine(…)`** functions
  - one supports time interpolation, the other does not
  - ops using time interpolation can be mixed with those that do not
- ***RefineAlgorithm*** has several **`createSchedule(…)`** functions
  - these are distinguished by level and hierarchy arguments
  - a ***RefineAlgorithm*** object can be used in different ways by creating different ***RefineSchedules***

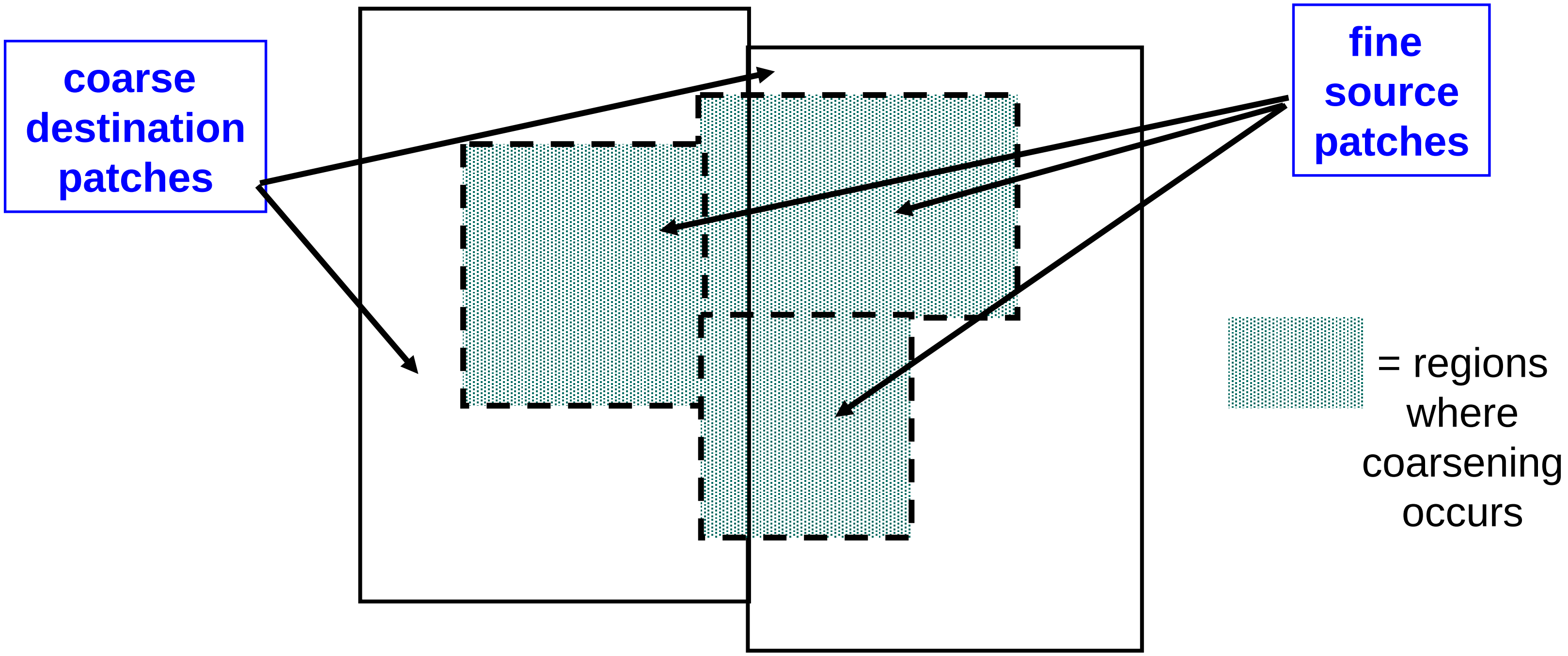
Notes on using refine algorithms and refine schedules II

- User-defined data refinement operations and physical boundary conditions are supported by passing a ***RefinePatchStrategy*** object to a **`createSchedule(…)`** function
- A ***RefineSchedule*** can be used repeatedly to transfer data as long as the patches involved in data movement are unchanged. Once patches change, the schedule must be regenerated.

SAMRAI “Coarsen Algorithm” and “Coarsen Schedule”

Coarsen Algorithm manages a data coarsen phase of computation

For example, fine mesh values may be averaged to a coarser level mesh for numerical consistency.


coarse destination patches
fine source patches
▨ = regions where coarsening occurs

SAMRAI framework supports data coarsening involving arbitrary combinations of variable quantities and operations within a single data transfer.

Using *CoarsenAlgorithm*, *CoarsenSchedule* to coarsen data on a patch hierarchy

1. Create a ***CoarsenAlgorithm*** object
2. Register data coarsen operations with ***CoarsenAlgorithm***
   - specify source and destination patch data indices
   - specify spatial coarsening operators
3. After all transfer operations are registered, create a ***RefineSchedule*** object
   - ***CoarsenSchedule*** depends on ***CoarsenAlgorithm*** object and patch hierarchy configuration
   - a ***CoarsenPatchStrategy*** object is needed for user-defined coarsening operations
4. Invoke the ***CoarsenSchedule*** to perform data coarsening operations

Notes on using coarsen algorithms and coarsen schedules

- ***CoarsenAlgorithm/Schedule*** objects are used only to coarsen data between two levels (fine to coarse) that may or may not reside in the same patch hierarchy.
- Typical coarsen operations do not involve data outside of the domain of the finer level. However, SAMRAI supports more complex operations when a larger “stencil” is required.
- ***CoarsenAlgorithm*** has **`registerRefine(…)`** function
- ***CoarsenAlgorithm*** has one **`createSchedule(…)`** function
- User-defined data coarsening operations are supported by passing a ***CoarsenPatchStrategy*** object to a **`createSchedule(…)`** function
- Once a ***CoarsenSchedule*** is created, it can be used repeatedly to coarsen data as long as the patches involved in the data movement are unchanged. Once patches change, schedule must be regenerated.

Topics to be covered in future

- Grid Geometry and Patch Geometry (Index space operations vs. coordinate system operations)
- Adaptive meshing operations
  - patch hierarchy construction and remeshing
  - error estimation
  - load balancing
- Input files and input database
- Restart files and restart manager
- Algorithm capabilities
- Solver interfaces
  - SAMRAI vector
  - vector interfaces to solver libraries
  - C++ wrappers for solver libraries

Topics to be covered in future ctd…

- Visualization files and tools
  - Vizamrai
  - VisIt
- Specialization and enhancement of SAMRAI capabilities
  - adding new patch data types
  - adding new grid geometry
  - etc.