Why DSLs Are Desirable

- **There is a fundamental tension between code maintainability, portability and performance**

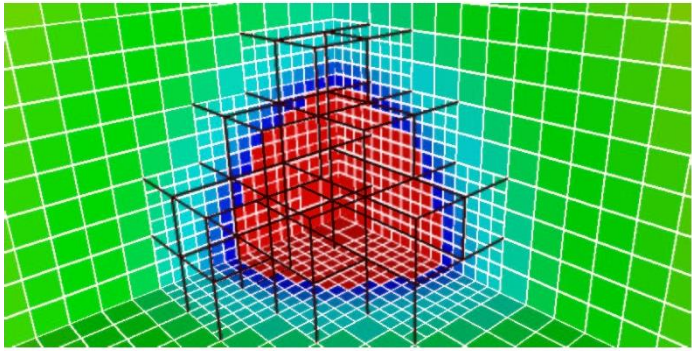
- **Heterogeneity in hardware architecture exacerbates the problem**
  - Nearly impossible to write code that is portable across platforms in current high level languages that also performs well everywhere
  - The only option is code transformation to retain portability

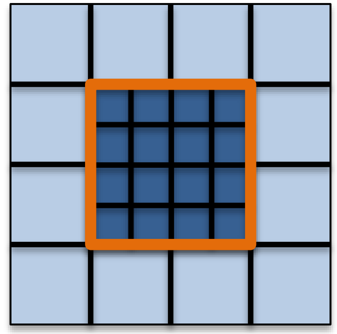
- **Urgent need for abstraction in programming model between high-level math and currently available languages**

Why Are Embedded DSLs Attractive to Applications

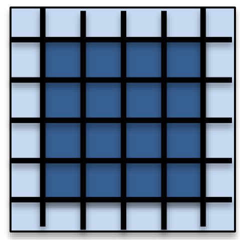
- **Often scientists inadvertently write code with optimization blockers**
  - Typical scientist coders not conversant with constraints of the compiler optimization
    - Compiler optimizations are by definition conservative: when in doubt don't optimize
  - Richer constructs help translation of algorithm into code without optimization blockers
  - if you make the problem easier for the compiler, you have a fighting chance to get good code.
- **Boutique solutions do not translate into production grade software**
  - Scientists are reluctant to add dependencies which have difficulty getting on new platforms
- **Applications won't use a language unless its longevity is guaranteed**
  - Enhancements embedded within an existing language make longevity more likely

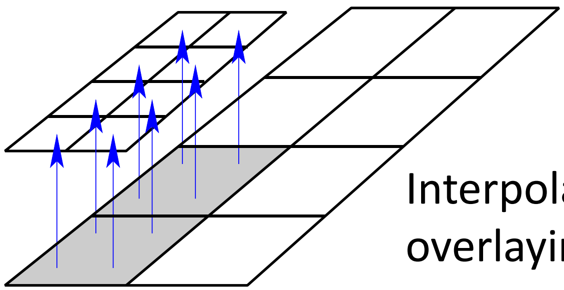
Applying a two-level AMR Operator



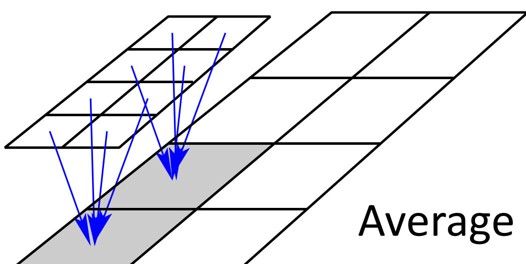


- Apply operator on the coarse grids
- Save fluxes at coarse-fine boundaries
- Fill ghost cells on the fine grids
- Apply operator on fine grids
- Increment fluxes at coarse-fine boundaries
- Apply flux correction at fine-coarse boundaries

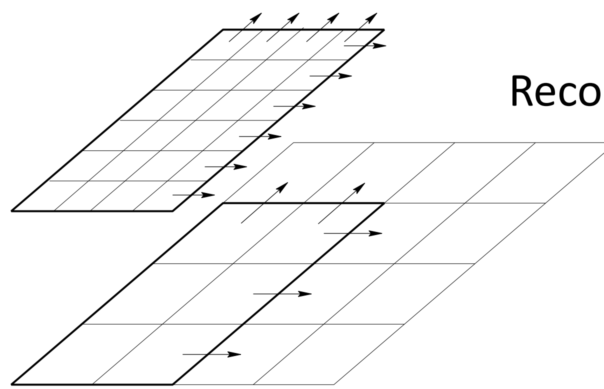




Interpolation from overlaying coarse cells



Average from fine cells

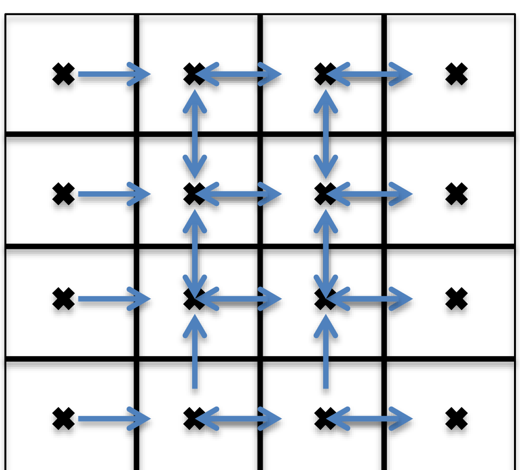


Reconcile fluxes

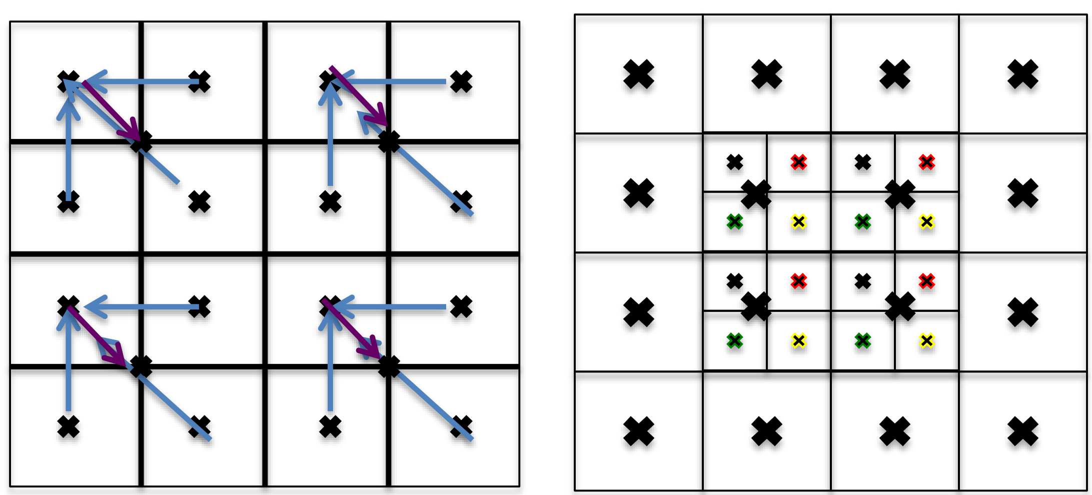
Basis for an EDSL - AMR Shift Calculus

- **A stencil operator is a sum of shifts multiplied by corresponding coefficients**
  - Offset specified by the shift relative to the target
    - No explicit ijk indexing (dimension independent code)
  - Shifts don't say where they are applied
  - The coefficients could be scalars or tensors
  - + and * operators for adding and composing stencils
- **Applying the stencil operator**
  - Weighted sum of some points on the mesh
  - Support nested hierarchies (example in Dan's talk)
- **Stencils are known at compile time, where to apply them is specified at run time**
  - Provides rich set of opportunities for compiler optimizations provided there is suitable runtime support

Level Shift



Shifts between Levels



Conclusions

- The code is written in higher level semantics –more opportunities for optimization
  - Functional dependencies articulated through composition of stencils
    - Possible to fuse procedures knowing the stencil composition
  - Spatial component expressed through the source and destination points
    - Possible to do custom decomposition/coalition of space depending upon the target architecture
    - Also possible to do over-decomposition to exploit pipelining potential through runtime management

- Having an embedded DSL useful
  - A very small API for compilers/code translation tools to work with
  - Flexibility of high level language for the complex logic of composition
    - Also for parts of the algorithm that do not map to shift calculus

Extra slides

Operations defined on Stencils

- **(S1+S2) =>union(S1,S2); coefficients of common shifts get added**
    - S1=<1,0|C1>,<0,0|C2>, S2=<0,0|C3>,<0,-1|C4>
    - S1+S2= <1,0|C1>,<0,0|C2+C3>,<0,-1|C4>
  - Defined for Level S1 and S2
- **(S1*S2) =>convolve(S1,S2); Shifts get added, coefficients get multiplied**
    - S1=<1,0|C1>,<0,0|C2>, S2=<0,0|C3>,<0,-1|C4>
    - S1*S2=<1,0|C1*C3>, <1,-1|C1*C4>,<0,0|C2*C3>,<0,-1|C2*C4>
  - Defined when
    - S1 and S2 of the same type (Level, CtoF or FtoC)
    - One of S1 and S2 is Level and the other is a half shift or multilevel shift