BERKELEY LAB

LAWRENCE BERKELEY NATIONAL LABORATORY

U.S. DEPARTMENT OF ENERGY

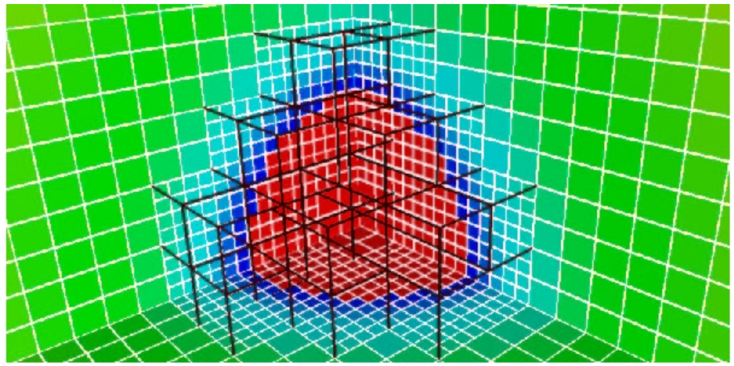
# The Chombo AMR Framework: Refactoring using AMRStencil as Defensive Programming

Brian Van Straalen

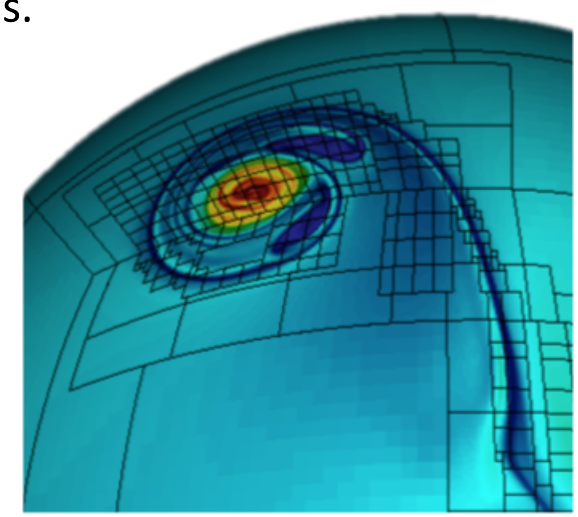
2016 Berkeley C++ Summit

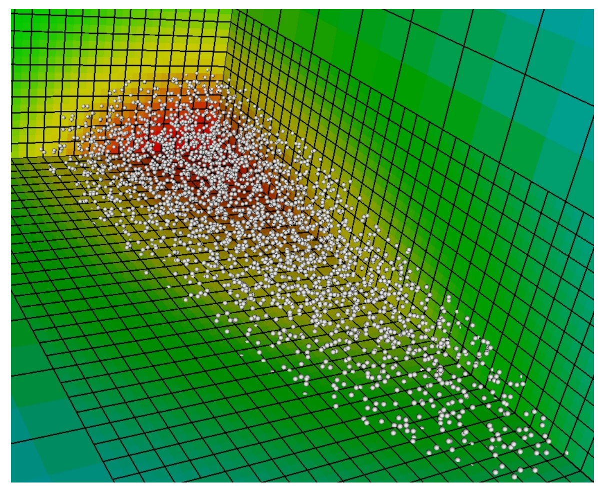
# Block-Structured Adaptive Mesh Refinement (AMR)

- Refined regions are organized into rectangular patches.



- Refinement in time as well as in space for time-dependent problems.
- Local refinement can be applied to any structured-grid data, such as bin-sorted particles.





# Chombo: AMR Software Framework

- **Goal**: to support a wide variety of applications that use AMR by means of a common software framework. Refactoring of **BoxLib** to better enable general usage patterns, such as support for embedded boundary representations of complex geometries.
- **Approach**:
  - Mixed-language programming: C++ for high-level abstractions, Fortran for calculations on rectangular patches.
    - SPMD Distributed memory parallelism, kernels written in Fortran (yuck).
  - Re-useable components, based on mapping of mathematical abstractions to classes. Components are assembled in different ways to implement different applications capabilities.
  - Layered architecture, that hides different levels of detail behind interfaces.
  - Significant effort expended in maintaining professional software development team responsive to a variety of users.

- **Status**: Chombo 3.2 Open Source Release – March, 2013.
  - Chombo 3.3 slated for Q1 2017
  - Currently 1M LOC

# The Problem

- Traditional HPC programming approach: simple programming model.
  - Serial programming on a node
    - vendor / third party compilers. Fortran has been our multidimensional array DSL
  - Bulk-synchronous SPMD programming across nodes
    - MPI Message passing
    - source-level frameworks to hide details of parallelism.
- Simple, long run of success. But, it will not deliver high performance on low-power HPC node architectures of the future at all scales.
  - Finer-grain parallelism
    - library overheads unacceptable,
    - heterogeneity becomes more of a problem.
  - Hierarchical parallelism / co-processing
    - NUMA on a node.
    - Different vendors may require different programming models, applications implementations.
  - Data movement is more expensive: both algorithms and software will need to change to perform more floating point operations per byte read / written.
- The programming model to address these concerns has not been worked out yet, but I need to start refactoring Chombo now….

# AMRStencil as a Defensive Programming Model

- AMRStencil: a limited embedded C++11 DSL
  - Main abstractions
    - Stencil, forall, Box, RectMDArray, LevelData
- Abstract away from Chombo Library and Applications:
  - Data placement
  - Dereferencing
  - Iteration
- Allow for multiple parallel execution models to be explored and compared across architectures.

# Multidimensional Arrays

Math:

C++:

Tensor-Valued Arrays on Boxes:

$$B \subset \mathbb{Z}^D$$
$$\phi : B \to \mathbb{T}^C \times \mathbb{T}^D \times \mathbb{T}^E$$

```
RectMDArray<double,CNUM> dirFluxes(ghostBox);
RectMDArray<double> LaplacianTemp(validBox);
RectMDArray<double,DIM,DIM> gradu(ghostBox);
RectMDArray<double,DIM,DIM,DIM> D2vel(validBox);
RectMDArray<double,DIM> divTau(validBox);
```

Slices (alias):

$$U : B \to \mathbb{R}^M$$
$$slice(U, 1, (q, q)) : B \to \mathbb{R}$$
$$slice(U, 1, (q, q))_i = U_i(q)$$

```
RectMDArray<double> velcomp =slice<double,DIM,1>
        (vel,Interval(veldir,veldir));
```

Pointwise operators:

$$\phi : B \to \mathbb{R}^C \times \mathbb{R}^E$$
$$f : \mathbb{R}^C \times \mathbb{R}^E \to \mathbb{R}$$
$$f@(\phi) : B \to \mathbb{R} \;,\; f@(\phi)_i = f(\phi_i)$$

```
forall(fOfPhi,phi,f,B);
```

Array algebra:

```
phi3 = phi1 + phi2; /// etc.
```

# Shifts and Stencils

Shifts:

$$\mathcal{S} = (\mathcal{S}_0, \ldots, \mathcal{S}_{D-1})\ ,\ \mathcal{S}_d(\boldsymbol{i}) = \boldsymbol{i} + \boldsymbol{e}^d$$

Stencil:

$$L = \frac{1}{(\Delta x)^2} \sum_{d=0}^{\mathbf{D-1}} \mathcal{S}^{\boldsymbol{e}^d} - 2\mathcal{I} + \mathcal{S}^{-(\boldsymbol{e}^d)}$$

Stencils are not quite first class objects: they must be fully specified at compile time in order to be efficiently applied as operators on arrays (see below).

Stencils have well-defined symbolic calculus
S1(S2(A)) = (S1*S2)(A)
S1(c1*A)+S2(A) = (c1*S1+S2)(A)

C++:

```
array<Shift,DIM> S=getShiftVec();

  Stencil<double> laplace =
   (-2.*DIM)*(S^zero);
  for (int dir=0;dir<DIM;dir++)
    {
      Point thishft=getUnitv(dir);
      laplace = laplace +
                  (S^thishft);
      laplace = laplace +
          (S^(thishft*(-1)));
    }
    laplace *= (1.0/a_dx/a_dx);
}
```

# Applying Stencils to Arrays

Math:

Single-level Stencil:

$$A = L(\phi) \textbf{ on } B$$

$$A_{\boldsymbol{i}} = \sum a_{\boldsymbol{s}} \phi_{\boldsymbol{i}+\boldsymbol{s}} \ , \ \boldsymbol{i} \in B$$

Fine to Coarse:

$$\mathcal{P}^{fc*}(U)_{\boldsymbol{i}} = U_{\lfloor \frac{\boldsymbol{i}}{\boldsymbol{r}} \rfloor}$$

$$\phi^c += \frac{1}{2^D} \mathcal{P}^{fc*}_{\boldsymbol{r}} (\mathcal{S}^{\boldsymbol{s}} \phi^f)$$

$$\phi^c_{\boldsymbol{i}} += \phi^f_{\boldsymbol{ri}+\boldsymbol{s}}$$

Coarse to Fine:

$$\mathcal{P}^{cf*}_{\boldsymbol{r}}(U)_{\boldsymbol{i}} = U_{\boldsymbol{ri}}$$

$$\mathcal{S}^{\boldsymbol{s}} \mathcal{P}^{cf*}_{\boldsymbol{r}} (\phi^f) += \phi^c$$

$$\phi^f_{\boldsymbol{ri}+\boldsymbol{s}} += \phi^c_{\boldsymbol{i}}$$

C++:

```
template <class T, int N, int D>
class IplusLaplacian : public Laplacian<T,N,D>

template <class T, int R>
class restrictSten : public Stencil<T,R,1>

template <class T, int R>
class prolongSten : public Stencil<T, 1, R>

RectMDArray<double,1> A = Laplacian<double>(Phi, B);

PhiC = restrictSten<double, 2>(PhiF, BF);

PhiF = prolongSten<double, 2>(PhiC, BC);
```

# Pointwise Functions: forall

- Not a new idea. Visit every point in `a_box`

```
template<class T, unsigned int Cdest, unsigned int
  Csrc, typename Func>
void forall(RectMDArray<T,Cdest>& a_dest,
     const RectMDArray<T,Csrc>& a_src,
     const Func& F, const Box& a_box);
```

- F can be any function that matches the `operator[]` signature of `a_dest` and `a_src`
- Did this the first time with std::bind a lot, now I use lambdas
- Next version of RectMDArray is variadic
  - I don’t quite know how to write forall with multiple variadic arguments

# Unions of Rectangles (Straight Out of Chombo)

Math:

Union of rectangles (fixed size, defined by bitmap). Syntax for domain boundaries suppressed.

$$\Omega = \bigcup_{\boldsymbol{i} \in \mathcal{B}} \Omega_{\boldsymbol{i}}$$

$$\Omega_{\boldsymbol{i}} = [\boldsymbol{i} b_{size}, (\boldsymbol{i} + \boldsymbol{u}) b_{size} - \boldsymbol{u}] \ , \ \boldsymbol{u} = (1, 1, \dots, 1)$$

Arrays defined over unions of rectangles:

$$\phi_\Omega = \{\phi : \Omega_{\boldsymbol{i}} \rightarrow \mathbb{R} : \boldsymbol{i} \in \mathcal{B}\}$$

Iterators: assumed to parallel, asynchronous.

Non-blocking communication between data over unions of rectangles.

C++:

```
BoxLayout(unsigned int a_N,
          vector<Point> a_points);
```

```
BoxLayout bl(s_numblockpower,s_points);
LevelData<double, 1> phi(bl, s_nghost);
```

```
for(BLIterator blit(layout);
   blit != blit.end(); ++blit)
  {initialize(phi[*blit]);}
```

```
   phi.exchange();
   phi.copyTo(phi2);
```

# Putting more together: Euler Class excerpts

Tensor type that matches state vector k

inline class member function WToFd

signature for our RK4 integrator

forall Boxes

- local temporary
- apply Stencil

- forall points
  - grab member function with lambda

```
typedef Tensor<double,DIM+2> V;

class Euler{
 inline unsigned int WToFd(V& a_F, const V& a_W,
       int a_d, double gamma);

 void operator()(EulerData& a_k, double a_time, double
   a_dt)
 {
 .
  for(BLIterator BL(a_k.state); BL.ok(); BL++) {
    Box B1 = grow(B0,2);
    RectMDArray<double,DIM+2> W_f(B1);
    W_f |= IplusLaplacian<double, -1, 24>(W, B1);

   forall(F_ave, W_f,
   [this, d, gamma](V& a, const V& b){ return
   WFtoFd(a,b,d,gamma);},
   B_FtoD);
   .
   .}
 }
};
```

# What is being gained here?

- I can refactor Chombo once. Starting now.
- No parallelism specified
- I can have multiple implementations of AMRStencil
  - serial C++11: written.  4500 LOC
  - UPC++: prototyped
  - MPI distributed memory : prototyped
  - OpenMP non-hierarchal: prototyped (caveat later)
  - Nested OpenMP: prototyped
  - Kokkos: Protonu Basu trying it out
- I can drop instrumentation in at the same time

## 2.2 GHz i7 processor g++ 5.3

```
-----------
Timer report 0 (42 timers)
--------------
[0] root 15.10033 100.0% 1 15691375104 1039.1 MFlops
   [1] Euler::advance 14.83863 98.3% 3600 15691276800 1057.5 MFlops
      [2] EulerOp::operator 14.17020 93.8% 14400 14544806400 1026.4 MFlops
         [3] EulerOp::operator::F_ave 5.14908 34.1% 28800 6559488000 1273.9 MFlops
            [4] Stencil::apply 3.43652 22.8% 57600 2156544000  627.5 MFlops
            [15] forall_RectMDArray_2 0.90566  6.0% 28800 1437696000 1587.5 MFlops
            [22] RectMDArray::plus 0.48670  3.2% 57600 1976832000 4061.7 MFlops
            [24] RectMDArray::operator*=::scalar 0.29600  2.0% 57600 988416000 3339.2 MFlops
         [5] EulerOp::operator::W_ave_f 1.59491 10.6% 28800 988416000  619.7 MFlops
            [6] Stencil::apply 1.59295 10.5% 28800 988416000  620.5 MFlops
         [7] EulerOp::operator::W_ave 1.27023  8.4% 14400 1270080000  999.9 MFlops
            [12] Stencil::apply 1.10959  7.3% 14400 705600000  635.9 MFlops
            [29] RectMDArray::plus 0.15890  1.1% 14400 564480000 3552.4 MFlops
         [8] EulerOp::operator::W_f 1.19174  7.9% 28800 718848000  603.2 MFlops
            [9] Stencil::apply 1.18962  7.9% 28800 718848000  604.3 MFlops
         [10] EulerOp::operator::U 1.11973  7.4% 14400 705600000  630.2 MFlops
            [11] Stencil::apply 1.11838  7.4% 14400 705600000  630.9 MFlops
         [13] EulerOp::operator::F_bar_f 0.93524  6.2% 28800 1482624000 1585.3 MFlops
            [14] forall_RectMDArray_2 0.93301  6.2% 28800 1482624000 1589.1 MFlops
         [16] EulerOp::operator::minusDivF 0.88386  5.9% 43200 770457600  871.7 MFlops
```

```
-----------
Timer report 0 (146 timers)
--------------
[0] root 1.21672 100.0% 1 2337881308 1921.5 MFlops
   [1] Multigrid::vCycle 1.09095 89.7% 9 2173254876 1992.1 MFlops
     [2] Multigrid::pointRelax 0.83451 68.6% 18 1755316224 2103.4 MFlops
       [3] Stencil::apply 0.70748 58.1% 1728 1585446912 2241.0 MFlops
       [10] LevelData::exchange 0.05280  4.3% 108 0    0.0 MFlops
          [12] RectMDArray::copyTo 0.05065  4.2% 22464 0    0.0 MFlops
       [14] RectMDArray::plus 0.02712  2.2% 864 56623104 2087.7 Mflops
       [16] RectMDArray::minus 0.02283  1.9% 864 56623104 2480.4 MFlops
       [17] RectMDArray::operator*=::scalar 0.02236  1.8% 1728 56623104 2532.3MFlops
     [4] Multigrid::vCycle 0.17891 14.7% 9 271662300 1518.4 MFlops
       [5] Multigrid::pointRelax 0.12692 10.4% 18 219414528 1728.7 MFlops
         [6] Stencil::apply 0.10379  8.5% 1728 198180864 1909.4 MFlops
         [20] LevelData::exchange 0.01313  1.1% 108 0    0.0 Mflops
       [22] RectMDArray::copyTo 0.01122  0.9% 22464 0    0.0 MFlops
        [32] RectMDArray::plus 0.00473  0.4% 864 7077888 1495.7 MFlops
        [43] RectMDArray::operator*=::scalar 0.00239  0.2% 1728 7077888 2966.6MFlops
        [45] RectMDArray::minus 0.00219  0.2% 864 7077888 3228.1 MFlops
        [13] Multigrid::vCycle 0.03951  3.2% 9 33963228  859.5 MFlops
          [15] Multigrid::pointRelax 0.02476  2.0% 18 27426816 1107.9 MFlops
```

# How is your sneaky code counting flops?

```
inline unsigned int EulerOp::WToFd(V& a_F, const V& a_W, int a_d, double a_gamma)
    const
{
  double F0 = a_W(a_d+1)*a_W(0);
  double W2 = 0.0;

  a_F(0) = F0;

  for (int d = 1; d <= DIM; d++)
  {
    double Wd = a_W(d);

    a_F(d) = Wd*F0;
    W2 += Wd*Wd;
  }

  a_F(a_d+1) += a_W(DIM+1);

  a_F(DIM+1) = a_gamma/(a_gamma - 1.0) * a_W(a_d+1) * a_W(DIM+1) + 0.5 * F0 * W2;

  return 2*DIM+8;
}
```

Currently, I'm cheating!
…but this is not a grand cheat
return value can be generated with static analysis

# Summary

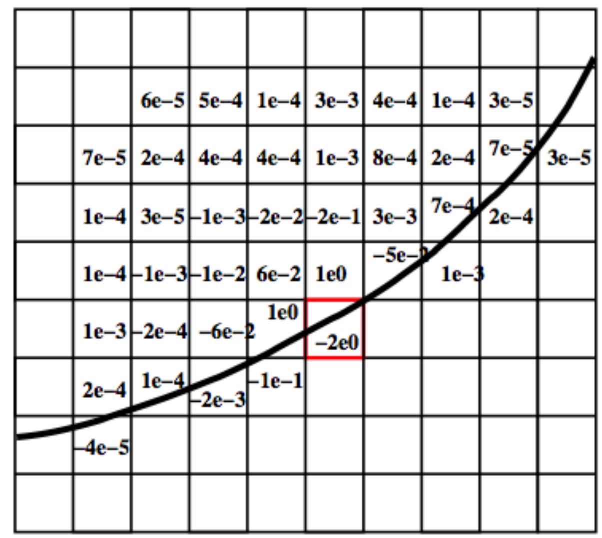
- Core execution distilled to agile core functionality in AMRSTencil
- The 1M LOC in Chombo can be programming model/ execution model agnostic
  - Separation of Concerns
  - Decent C++ programmers can use AMRStencil
  - Performance Engineers can try new stuff without learning all of Chombo
- BLIterator more flexible than it looks
  - order not specified
  - BLIterator::Box can be tiled, or threaded
  - Charm++ version of BLIterator (Phil Miller)

# Flies in the Ointment

- …I can *almost* make Stencils constexpr

- Not every Stencil is knowable at compile-time
  - Embedded Boundary Chombo
    - Stencil points and weights from least-squares solve
  - Currently using runtime stencil playback

- MPI and UPC++ init.
  - AMRStencil needs init
- OpenMP thread model
  - BLIterator can't launch omp parallel
    - trapped by basic block scope
  - reduction is done with directive, not runtime
  - programming model escaping refactoring




(d) Stencil for a cut cell using weighted least squares.

# What Haven't we Discussed?

- Translation technologies
  - AMRStencil was originally a target for ROSE DSL research. Much care to distinguish compile time vs runtime semantics
  - How much parsing do you really need?
    - full C++11 AST ?
- Tuning
  - Yes, you can implement AMRStencil with meta-programming, but can you tune it?
    - Basu, P., Hall, M., Williams, S., Van Straalen, B., Oliker, L., & Colella, P. (2015, May). Compiler-Directed Transformation for Higher-Order Stencils. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International* (pp. 313-323). IEEE.