Using POOMA Particles Classes

PARALLEL **O**BJECT-**O**RIENTED **M**ETHODS AND **A**PPLICATIONS

Timothy J. Williams
CIC-19, Los Alamos National Laboratory

Tecolote Class
LANL
August 25, 1998

Outline

- POOMA and particles
- Particle attributes
- Managing a set of particles
- Particle layout, loading, & boundary conditions
- Particle-mesh interaction
- Particle load balancing
- Particle-particle interaction
- I/O and visualization
- Example

Other POOMA Particles Documentation

- POOMA User Guide, especially Chapter 7
  - http://www.acl.lanl.gov/pooma/documentation/userguide
- Header files
  - `$POOMA_ROOT/src/Particles/*.h`
- Test programs
  - `$POOMA_ROOT/src/test/particle/*.h`
- External data connection (I/O, visualization, PAWS)
  - http://www.acl.lanl.gov/pooma/documentation/externGuide.pdf
- These viewgraphs:
  - http://www.acl.lanl.gov/pooma/papers/TecClassPOOMAParticles

Applications Drive POOMA Particles Design

- ER (HPCC Grand Challenge Applications)
  - Numerical Tokamak Turbulence Project (PIC)
  - Computational Accelerator Physics (PIC)
- ASCI
  - MC++ (Monte Carlo neutron transport)
- Additional projects leverage previous work
- Additional projects drive improvements in POOMA

Particle Attributes

- `template <class T> class ParticleAttrib`
  - Value of type `T` for every particle
    - Scalar (`double`,….)
    - `Vektor<double,3>, Tenzor<double,3>,...`
  - Analogous to `Field<T,….>`
  - Distributed object
  - Data parallel and single-particle operations
    - Use expression templates

Data-Parallel Math On `ParticleAttrib`

- All particles

```
ParticleAttrib<double,3> mu, rho, w, speed, mass;
ParticleAttrib<Vektor<double,3>> v;
double scaleFactor;
mu = 2.0*rho*w;
mu *= scaleFactor;
speed = sqrt(pow(v(0),2) + pow(v(1),2) +
             pow(v(2),2));  // note component access
```

- Reductions

```
Vektor<double,3> totMom = sum(mass*v);
```

- Conditionals

```
rho = where((rho > 1.0e-8), rho, 0.0);
```

Single-Particle Math On `ParticleAttrib`

- When *not* to use data-parallel?
  - Many particle-dependent temporaries (⟶`ParticleAttrib` temporaries)
  - Complicated particle-dependent logic
- Single-particle indexing, using `[int]`

```
for (int i=0; i<nLocalParticles; i++) {
  mu[i] = 2.0*rho[i]*w[i];
  if (mu[i] < 1.0e-8) mu[i] = 0.0;
}
```

  - SPMD

Managing A Set Of Particles

- Abstract base class
  `template<class PLayout> class ParticleBase`
  - Template parameter `PLayout`
    - Must inherit from `ParticleLayout` base class
    - Contains spatial dimensionality
    - Manages processor ownership of particles
- User derives custom particles class

Key Members Of `ParticleBase`

- Two `ParticleAttrib` data members
  - `R` - position (`Vektor<T,Dim>`)
  - `ID` - global particle identifier (`int`)
- Member functions
  - `create()`, `globalCreate()`, `destroy()`
    - create and destroy particles
  - `update()`
    - manage memory
    - move particles between processors
  - `getBConds()`
    - Get reference to `ParticleBConds` object
- `Typedef`s, such as `ParticlePos_t`

Example Custom Particles Class: `Tracers`

```
class Tracers : public
ParticleBase<ParticleSpatialLayout<double,3>> {
  public:
  // Attributes (besides R and ID)
  ParticleAttrib<Vektor<double,3>> v;  // velocity
  // ...other attributes would go here ....
  // constructor
  Particles(Layout_t *L) :
    ParticleBase<Layout_t>(L) {
     addAttribute(v); // add attribute to base's list
     // other addAttribute() calls....
  }
};
```

Particle Layout Classes

- Manage processor ownership of particles

```
template <class T, unsigned Dim>
  class ParticleUniformLayout
```

  - Divide particles evenly among processors

```
template <class T, unsigned Dim, class
  Mesh> class ParticleSpatialLayout
```

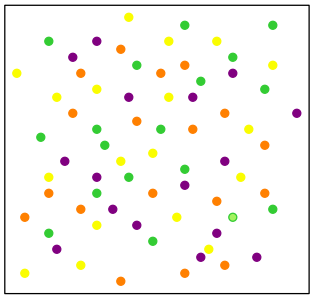
  - Spatial subdomains
  - PE owning subdomain owns all particles in it

```
template <class T, unsigned Dim, class
  Mesh> class ParticleInteractLayout
```

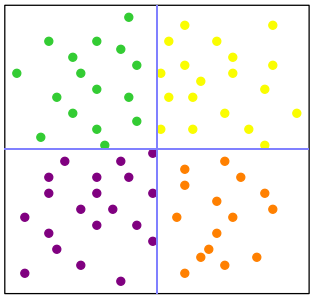
  - Adds mechanisms for particle-particle interaction

Particle Layout Classes (cont'd)

`ParticleUniformLayout`



`ParticleSpatialLayout`



Processor 0,1,2,3

ParticleSpatialLayout Construction

```
unsigned Dim = 3;
typedef UniformCartesian<Dim> M;
int nVertices[Dim]; // Must initialize (not shown)!
NDIndex<Dim> allVerts;
for (int d=0; d<Dim; d++)
  allVerts[d] = Index(nVertices[d]);
Vektor<double,D> origin, meshSpacings; //initialize!
M mesh(allVerts, origin, meshSpacings);
CenteredFieldLayout<Dim,M,Cell> fl(mesh);

ParticleSpatialLayout<double,Dim,M>
  psl(fl, mesh);
```


Particle Boundary Conditions

- `ParticleBconds` class
  - What happens to particles that cross global boundaries
  - Contains container of function pointers

    ```
    typedef T (*ParticleBCond)(const T, const T, const T);
    ParticleBCond BCList[2*Dim];
    ```

    - Act on single particle position in one direction
  - Invoked on all particles by `update()`
- POOMA provides canned functions
- User may supply custom functions

POOMA-Provided Particle Boundary Conditions

- Periodic BC

```
template<class T>
T ParticlePeriodicBCond(const T t, const T minval,
                        const T maxval)
```

- Reflecting: particles bounce back at boundary:

  `T ParticleReflectiveBCond(`*...same args...*`)`

- Particles stick to face

  `T ParticleSinkBCond(`*...same args...*`)`

- Null (default for all boundaries)

  `T ParticleNoBCond(`*...same args...*`)`

Particle Boundary Conditions Example

```
typedef UniformCartesian<2> M;
M mesh(....);
CenteredFieldLayout<2,M,Cell> fl(mesh);
typedef ParticleSpatialLayout<double,2,M> PL;
PL pl(fl, mesh);
Tracers<PL> tracers(&pl);
tracers.getBConds()[0] = ParticlePeriodicBCond;
tracers.getBConds()[1] = ParticlePeriodicBCond;
tracers.getBConds()[2] = ParticleReflectiveBCond;
tracers.getBConds()[3] = ParticleReflectiveBCond;
```

Particle Loading

1. Construct user particles object (e.g. `Tracer` instance)
2. Create particles
   - `create(npLocal)` - SPMD PE-local creation
   - `globalCreate(npTot)` - divides `npTot`/#PEs
3. Assign initial values to `ParticleAttrib`s
   - read input
   - data-parallel/single-particle expressions
4. `Tracer::update()`
   - set up proper processor ownership
   - consults `PLayout`

Particle-Mesh Interaction

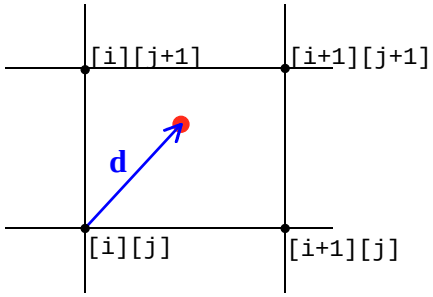
- `Interpolator` base class
  - Maps continuous position ⟷ mesh position
- Derived classes provide `scatter()`, `gather()`
  - Single particle ⟷ mesh neighborhood
  - POOMA implements `IntNGP`, `IntCIC`, `IntSUDS`
  - `Other interp. formulas: user-defined class`
- `ParticleAttrib` functions apply to all particles

```
template <unsigned Dim, class M, class C,
          class PT, class IntOp>
 void ParticleAttrib<T>::scatter
  (Field<T,Dim,M,C>&,
    const ParticleAttrib< Vektor<PT,Dim> >&,
    const IntOp&) const;
```

Particle-Mesh Interaction (cont'd)

- POOMA interpolators `IntNGP`, `IntCIC`, `IntSUDS`
  - `gather()`
    - Interpolate from mesh points to single particle position `R[i]`
    - Assign value to `pa[i]`
  - `scatter()`
    - Interpolate from single particle position to mesh nodes/cells/....
    - *Accumulate* value to field at node/cell-position
      - Really *scatter-add*

POOMA `IntNGP` Interpolator



- Particle k
- `ParticleAttrib<T> pa`
- `Field<T,2,M,Vert> f`
- Distance **d** = dx **x** + dy **y**

```
pa[k]=(1-dx)*(1-dy)*f[i][j] + dx*(1-dy)*f[i+1][j]
      (1-dx)*dy*f[i][j+1]   + dx*dy*f[i+1][j+1]
```

Particle-Mesh-SIndex Interaction

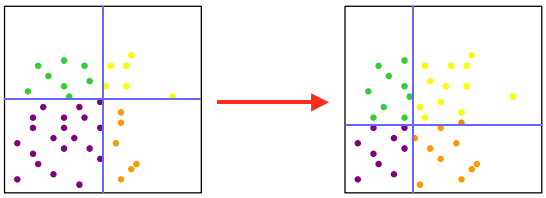
- Sparse index gather/scatter

```
Field<double,D> f(...), g(...), h(...);
ParticleAttrib<double> pa(...);
SIndex<D> S;
S = where(gt(f, 1.0e-6));
pa[S] = g[S] + h[S];
f[S] = pa[S]*g[S];
```


Particle Load Balancing

- Redistribute particles evenly among processors
- Function changes *layout* and all object using it

```
template <class T, unsigned Dim, class Mesh>
void BinaryRepartition
 (ParticleBase<ParticleSpatialLayout<T,Dim,Mesh>>& PB)
```



Processor 0,1,2,3

Particle-Particle Interaction

- Pair-list iterator over neighbor particles
  - Member of `ParticleLayout` class
  - Exposed for use by `ParticleBase` class
- `ParticleInteractLayout::pair_iterator`
  - All other particles within fixed radius of particle

    `ParticleInteractLayout::setInteractionRadius(radius);`
- Uses ghost particles

Particle-Particle Interaction Example

```
class Molecules:
         ParticleBase<ParticleInteractLayout<double,D>;
Molecules m(...);
m.getLayout().setInteractionRadius(4.0);
m::pair_iterator neighbor, neighborEnd;
for (int i=0; i < m.getLocalNum(); ++i) {
  m.getPairList(i, neighbor, neighborEnd);
  for ( ; neighbor != neighborEnd; ++neighbor) {
    Molecules::pair_t pairdata = *neighbor;
    int n = pairdata.first;  // Particle index of neighbor
    double sep2 = pairdata.second;     // (Nbr distance)^2
    m.R[i] += m.vel[n] * 2.0;          // Interact
  }
}
```

I/O

- Parallel File I/O: `DiscParticle` class
  - One file set `foo.{meta,data,config}` per SMP
  - Analogous to DiscField
  - Write/read single `ParticleAttribs` or entire `ParticleBase` objects

```
class Atoms:
  ParticleBase<ParticleSpatialLayout<double,3>>;
Atoms a(...);
DiscParticle dp("a","a.config",INPUT,"Atoms a");
dp.read(a);
```

  - External data connection (`DataConnect`) approach also works

Visualization

- External data connection to ACLVIS
- ACLVIS tool visualizes `ParticleAttrib` (in `ParticleBase`) data several ways:
  - Points, colored by scalar attribute value
  - Spheres, colored by scalar attribute value
    - Keep N timesteps and display as trace path
  - Vector arrows (for `ParticleAttrib<Vektor<T,D>>`)
    - Arrow length = vector magnitude
    - Keep N timesteps and display as trace path

Example: Tracer Particles (1/8)

```
#include "Pooma.h"
#include "Particle/IntNGP.h"

// ---------------------------------------------------------
// Custom particles class Tracers
template<class PL>
class Tracers : public ParticleBase<PL> {
public:
  PL::ParticlePos_t       v; // tracer particle velocity
  ParticleAttrib<double>  s; // tracer particle speed
  double mass;               // tracer particle mass (const)
  Tracers(PL* pl, double m) : ParticleBase<PL>(pl), mass(m) {
    addAttribute(v); // register the particle attributes....
    addAttribute(s); // register the particle attributes....
  }
};
// ---------------------------------------------------------
```


Example: Tracer Particles (2/8)

```
int main(int argc, char *argv[]){
  Pooma pooma(argc, argv);
  Inform pout(NULL,0);

  const unsigned Dim = 3;            // dimension of positions
  const int nx=32, ny=32, nz=32;     // grid sizes (cells)
  const unsigned int totalP = 5000;  // number of particles
  const int nt = 100;                // # of trace-timesteps
  const double dt = 0.1;             // tracer timestep
  const double mass = 0.1;           // tracer particle mass
  const double pi = acos(-1.0);
```

Example: Tracer Particles (3/8)

```
// Create mesh and FieldLayout objects
Index I1(nx+1), J1(ny+1), K1(nz+1);
typedef UniformCartesian<Dim> M;
M mesh(I1,J1,K1);
CenteredFieldLayout<Dim,M,Cell> layout(mesh);

// Velocity Field u(x,y,z). Define as a cylindrical vortex
// parallel to z, center in x-y, finite radius R0=8.0,
// speed = ((x,y) - (x0,y0))/R0
Field<Vektor<double,Dim>,Dim,M,Cell> u(mesh,layout); u = 0.0;
double R0 = 8.0;
Vektor<double,Dim> c; c[0] = nx/2-1; c[1] = ny/2-1; c[2] = nz/2-1;
// Get Field of cell-center positions:
Field<Vektor<double,Dim>,Dim,M,Cell> x(mesh,layout);
x = mesh.getCellPositionField(x);
```


Example: Tracer Particles (4/8)

```
// Do the trig and assign u:
Field<double,Dim,M,Cell> r(mesh,layout);
Field<double,Dim,M,Cell> theta(mesh,layout);
NDIndex<Dim> cells;
cells[0] = Index(nx); cells[1] = Index(ny); cells[2] = Index(nz);
r[cells] = sqrt((x[cells](0) - c[0])*(x[cells](0) - c[0]) +
        (x[cells](1) - c[1])*(x[cells](1) - c[1]));
theta[cells] = acos((x[cells](0) - c[0])/r[cells]);
theta[cells] = where(lt((x[cells](1) - c[1]),0.0),
             2.0*pi - theta[cells],
             theta[cells]);
assign(u[cells](0), where(le(r[cells], R0),
                   (r[cells]/R0)*sin(pi - theta[cells]),
                   0.0));
assign(u[cells](1), where(le(r[cells], R0),
                   (r[cells]/R0)*cos(pi - theta[cells]),
                   0.0));
```

Example: Tracer Particles (5/8)

```
// Create nearest-grid-point interpolater object
IntNGP interp;

// Create an empty Tracers object, set up reflecting BC's
typedef ParticleSpatialLayout<double,Dim> PSL;
PSL pLayout(layout);
Tracers<PSL> t(&pLayout, mass);
// 2*Dim faces to set boundary conditions on:
for (int face=0; face < 2*Dim; face++) {
  t.getBConds()[face] = ParticleReflectiveBCond;
}
```


Example: Tracer Particles (6/8)

```
// Initialize the particle object (create particles):
t.globalCreate(totalP);

// Random initial positions, uniform in box:
assign(t.R(0), PoomaRandom * nx);
assign(t.R(1), PoomaRandom * ny);
assign(t.R(2), PoomaRandom * nz);

// Redistribute particles based on spatial layout
t.update();
```

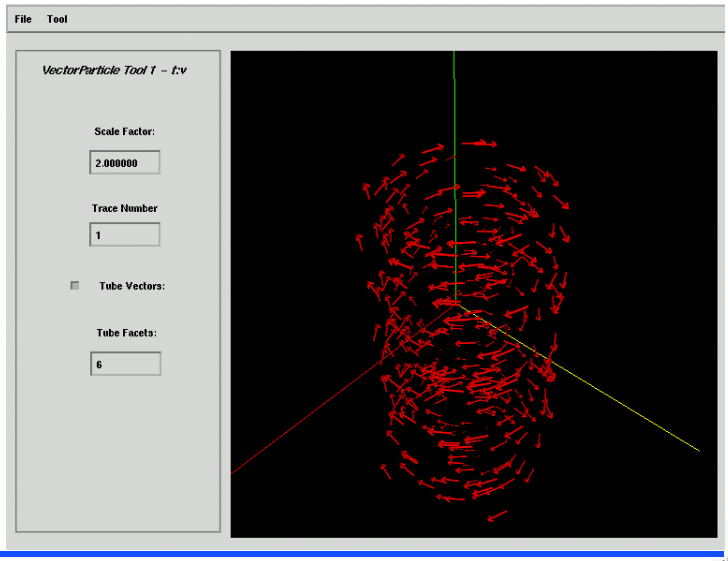
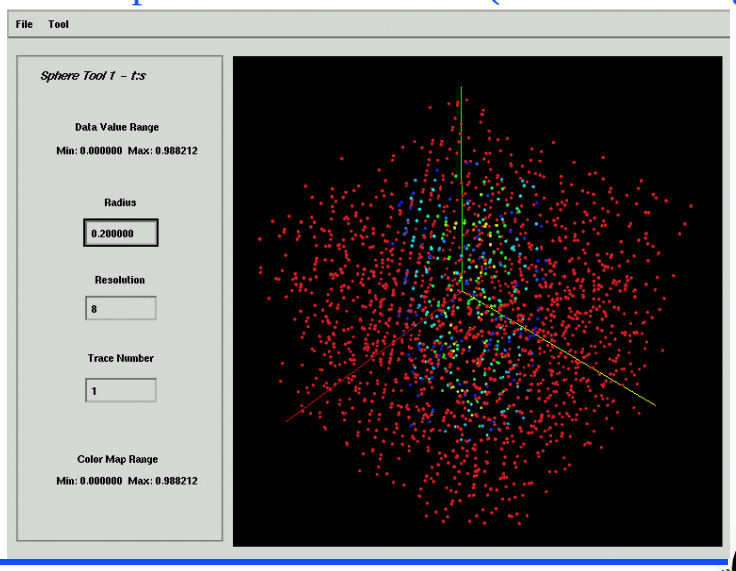
Example: Tracer Particles (7/8)

```
// Establish data connection (for ACLVIS)
DataConnect *dc = DataConnectCreator::create("tracers");
dc->connect("t", t);
dc->connect("R", t.R);
dc->connect("s", t.s);
dc->connect("v", t.v);
dc->connect("u", u);

// Update external connections, set interaction breakpoint
dc->updateConnections(); dc->interact();
```


Example: Tracer Particles (8/8)

```
// Begin main timestep loop--------------------------------
for (int it=0; it<nt; it++) {

  // Gather velocity from flow Field to particles:
  t.v.gather(u, t.R, interp);
  t.s = sqrt(dot(t.v, t.v)); // compute speed

  // Push the particles
  t.R += t.v*dt;

  // Redistribute particles based on spatial layout
  t.update();

  pout << "Finished iteration " << it << endl;
  dc->updateConnections(); dc->interact();
}

delete dc;
return 0;
}
```

Example: Tracer Particles (ACLVIS Image 1)



Example: Tracer Particles (ACLVIS Image 2)