PIC Modifications to AMRNodeElliptic

dbs

April 4, 2003

1 Introduction

This document describes the design and implementation of a particle-in-cell (PIC) functionality for Chombo's AMRNodeElliptic class, and the interactions and interfaces with applications that would use this functionality (in particular, the WARP and IMPACT physics codes).

The design of the interfaces and the internal Chombo implementation must allow the code to be executed in parallel. This assumes that the application code uses MPI and that all processors are active and interfacing to Chombo.

The essential steps in using the PIC capability are:

1. transfer particle location data from the application
2. generate an AMR mesh satisfying the max-particle-per-cell criterion
3. distribute particle charges to the grid nodes
4. solve a Poission problem at the grid nodes to get the potential (Φ)
5. compute $\vec{\mathbf{E}} = -\nabla\Phi$ at the grid nodes
6. interpolate $\vec{\mathbf{E}}$ back to the particle locations
7. transfer $\vec{\mathbf{E}}$ at the paticle locations to the application

See §4 for details.

The Chombo implementation will use Dan Martin's ParticleTools classes for managing the particle data on the grids.

2 Requirements for application interface

It is assumed that the applications will be Fortran codes. Thus is it desirable to keep the data interface simple. For this reason, the data structures used in the interface will be simple 1-d vectors or 2-d arrays. There will be no requirement for ordering of the particles in the vectors/arrays. This is the least-common-denominator interface. Higher-level interfaces can be built on this, to provide a simpler interface for more advanced applications and to allow for higher performance implementation strategies.

The primary functionality requirement for the application interface is for a routine that takes the particle positions as input and returns the electric field at those positions as output. This allows the application to use the `Chombo` functionality with minimal disturbance to the rest of the application code and algorithms.

An important issue in the design of interfaces is the parallel distribution of particles to processors. The simplest approach is to ignore the issue and let the `Chombo` code redistribute the particles however it requires. This may have unbearably bad performance. This can be optimized by adding constraints on the distribution used by the application. Distributions based on geometry will be sufficient for some applications (`WARP`and `mad9p`). Whether there are acceptable constraints that provide acceptable performance is not known.

Another approach is to let `Chombo` determine the distribution. This probably will complicate both the interface and the implementation. It also burdens the application with the task of distributing the particles to `Chombo`'s specification, and burdens both with maintaining the distribution as the particles move. A middle approach is to allow the application to specify the initial distribution, but let `Chombo` return the particles in a different distribution, and provide an interface for the application to explicitly define where particles move (and are added). Such an interface would be fairly complex, which is counter to our goal of keeping the interface simple for the Fortran applications, although it might be fine for C/C++ applications.

One or more secondary routines will be needed to specify various constants: scalar constants such as the charge on the particles (which is assumed to be constant on all particles for all time), the max number of particles in a cell, an error/convergence threshold, etc; and non-scalar data such as geometry and boundary conditions.

AMRNodeElliptic implements two Fortran routines to handle the geometry and boundary conditions: `reachablenodes()` specifies which nodes are outside the boundaries (domain or embedded) or on the other side of a split cell (thus not "reachable"); `nodalcoefficients()` computes the template coefficients of the discrete Laplacian, accounting for the boundaries. *[...I assume this code came from* `WARP`*– will we want to use it for* `IMPACT`*???]*

In the long run, we will want the application codes to use the same distribution of particles to processors that `Chombo` uses. Ideally using the same data structures, although this may be impractical for the Fortran applications.

2.1 interface to `IMPACT`

According to Qiang Ji[1], the `IMPACT` code stores particles as an array of structures containing the coordinates of the particle and its momentum. The field $\vec{\mathbf{E}}$ is not stored in the structure. This will require either a 2-d array interface for the coordinates or a 1-d interface that

[1]<JQiang@lbl.gov>

understands strides. I assume either type will be acceptable for returning the field values, as long as it is consistent.

Particles are distributed to processors using domain decomposition in the physical domain. *[...Ji hasn't explained yet how the subdomain boundaries are chosen...]*

2.2 interface to `WARP`

According to Dave Grote[2], the `WARP` program stores particles in 1-d arrays, with each coordinate component (x, y, z) in a separate array (likewise for components $\mathbf{E_i}$ of the field). Note that this ordering is the transpose of that used by `IMPACT`, so our interface will have to allow for both orderings.

The particles are not precisely ordered, although they are "mostly" ordered by the z coordinate, becoming less so as the computation progresses and particles move. This may be useful in optimizing for data locality, but can't be relied on to build an algorithm.

The distribution of particles to processors is precisely ordered by z. We can take advantage of this in the `Chombo` implementation.

2.3 interface to `mad9p`

Andreas Adelmann's[3] code `mad9p` is a C++ code that already uses the POOMA class library for solvers and particles. The current plan is to interface to `Chombo` directly by importing the `Chombo` classes into the application and calling AMRNodeElliptic. Andreas is willing to discuss letting `Chombo` define the distribution of particles. He currently does it using recursive bisection in 3-d. This presents a useful platform for prototyping interfaces, since it will be easier to do in C++ than Fortran, and Andreas is willing to do much of the coding.

[...should check out the Pooma *particle class to see if it has any useful or interesting features...]*

3 `Chombo` data structures for particles

Conceptually, we need to store the particles at each cell in the AMR grids. The data stored with each particle will be the coordinates (x, y, z) received from the application, the computed electric field $\vec{\mathbf{E}}$ that is returned to the application, and the original array index and processor number.[4] Since the interface requires $\vec{\mathbf{E}}$ to be returned in the same ordering as the particles, we'll need to store the array index so we can put the results back in the argument array(s) in the right places. The processor number is needed because the grid cell that logically "contains" a particle may not be on the processor that contains the input data for that particle.

[2] <DPGrote@lbl.gov>

[3] <aaadelmann@lbl.gov>

[4] This assumes that charge is constant; if this changes later the charge will have to be stored with the particle as well.

Dan Martin's `ParticleTools` library[5] will be used. A class `ChargedParticle`, derived from `BinItem`, will encapsulate the particle data (see §5.1). Particle data should be stored only on the finest level in the AMR hierarchy at every node. Initially, when the charge is distributed to the finest level nodes, it can be copied to coarser level nodes. Likewise, after the field is computed only the finest level nodes are needed to interpolate back to the particle locations. The data structure that stores particle information will have to be able to represent 3 states at any grid point: empty, not-empty, and not-finest.

4 `Chombo` algorithms for handling particles

4.1 transfer particle location data from the application

Ideally, this will not require much effort if the application's data structures can be used without being copied or rearranged. For example, `WARP` uses 1D arrays for everything, which can be passed to `Chombo` without any extra copying. This may require some new functionality in the ParticleTools classes, but nothing major (see §5.2 below).

It is conceivable that the particles will have to be sorted in order to access them efficiently. This could be expensive in the parallel case. For `WARP`, it can be assumed the particles are *almost* sorted. This might be used to optimize a sorting algorithm.

In parallel, this step overlaps with the next.

4.1.1 Initial implementation

The initial implementation will use separate 1-d arrays for all variables, along with a stride for each one. This will handle both the 1-d and 2-d cases, at the cost of a slightly ugly interface and a little extra coding.

4.2 generate mesh satisfying max-particle criterion

Generating a grid is straightforward in serial but gets complex in parallel.

In serial it will be a two stage process: first, choose a mesh spacing for the coarsest mesh and compute particle counts for each cell; second, tag any cell that has more than the maximum number of particles, merge these tags with any other tags (e.g. based on RHS or ∇RHS), and refine the mesh using the resulting tags. Recurse the second stage until no cells have too many particles and the desired mesh level is reached.

In serial, computing the number of particles per cell on a level is $O(bp)$, where b is the number of boxes on the level, and p is the number of particles. For small b, linear search through the boxes will suffice. For large b, it may be desirable to optimize this search. *[Ted has a function for this that may be reusable.]*

If the number of particles is small, this approach also can be used in parallel by gathering the input particle data from all processors into one, then generating the mesh. Dave Grote

[5] see $CHOMBO_HOME/lib/src/ParticleTools and $CHOMBO_HOME/ChomboDoc/ParticleTools

says he has plans for large simulations with a billion particles, so we'll need a parallel solution eventually.

In parallel, the complication arises because the input mapping of particles to processors is not regular, so the particles contained in a box may reside on multiple processors, and there's no way to know which *a priori*. Since `MeshRefine` is always serialized, some amount of information based on the particles will have to be gathered onto one processor no matter what algorithm is used.

To refine the initial coarsest grid, all that's really needed is the number of particles in each cell. For a single grid, it is simple to broadcast the grid description (the box, the physical coordinates of one corner and the mesh spacing, h) to all processors, let each processor compute the counts for its own particles, and gather/reduce the results back onto the processor doing the mesh refinement. This procedure can be repeated for all boxes on the finer level grids.

This approach will work, but has the disadvantage that it gets more expensive for large problems that have more grid points because you have to send more messages and the messages are larger. The obvious optimization of sending all the boxes on a level in a single message reduces the latency cost, but doesn't affect the bandwidth cost, which could be very large if the number of non-empty cells on each processor is high.

If the density of non-empty cells is low enough, data on an IntVectSet could be used instead of a BaseFab that represents the whole box. This would reduce the bandwidth required, but would add some extra (serial) computation to construct the IVS. A further optimization would be to request particle counts only for the cells that are over the max threshold in the current level. This would reduce the bandwidth required even further, but would add more serial computations, and would raise the problem of dealing with cells that are split between processors.

Redistributing the particles to processors such that there is no geometric overlap would eliminate the need for a reduction operation, which would reduce both the latency and bandwidth costs, but might create a significant overhead cost. Further restricting the distribution to be along coarse grid lines, so that no cell is split between processors, would help a lot.

Another problem that any algorithmic approach will have to face is handling particles that lie exactly on a grid line. The algorithm has to guarantee that only one cell counts that particle, even though there may be 2^{SPACEDIM} cells near the particle. Any tie-breaking rule must also be consistent for refined cells, and handle coarse-fine boundaries correctly. In parallel, there is the extra complication that the cells may be on different processors.

[It would be a good idea to run some synthetic examples and measure the actual costs involved before investing any effort in a better (more parallelized) algorithm.]

4.2.1 Initial implementation

The initial implementation will use the simple approach of broadcasting the grid boxes to all processors, and having each processor count the number of particles in every cell of every box. The counts are gatherer from all processors into one and tags are computed from the accumulated counts.

The algorithm is:

```
construct level 0 grid from problem domain and size parameters

for levels 0 to max_level

    broadcast boxes to all processors
    for each box
        count particles in each cell into a BaseFab<int>
        sum the BaseFab across processors into proc. 0
        in proc. 0, tag every cell where count > threshold
    generate boxes for next level using tags on this level (BRMeshRefine::regrid)
```

4.3 distribute particle charges to the grid nodes

Because multigrid uses residuals as the RHS on coarse levels, we only have to distribute charges to the nodes on the finest level and to coarse nodes on coarse-fine boundaries (since all coarse nodes are "valid").

Distributing charge from particles to the grid nodes will be done by interpolating using the product of 1-d hat functions. Charge is distributed only to the nodes surrounding the cell that contains the particle, not to neighboring cells. Let ρ^p be the charge for particle p at location X^p. Then $\Delta\rho_p^n$, the contribution to the charge at grid node n from particle p, can be computed using:

$$\Delta\rho_p^n = \rho^p \prod_{d=1}^{\text{SPACEDIM}} \frac{(1 - |\frac{X_d^n - X_d^p}{h_d}|)}{h_d},$$

where X^n is the location of node n and h_d is the mesh spacing in coordinate direction d.

There are several issues that arise here: scaling to account for grid cell size; scaling for the charge per particle; handling nodes at domain boundaries and coarse-fine boundaries between levels; ensuring charge conservation, projection from fine to coarse and vice versa.

Charge conservation is expressed by the equation:

$$\sum_{p=1}^{Np} \rho_p = \sum_{l=0}^{l_{max}} \sum_{i=1}^{N_l} \bar{\rho}_i * v_l$$

where $\bar{\rho}_i$ is the charge density at node i on level l, $v_l \equiv \prod_{d=1}^{\text{SPACEDIM}} h_d^l$ is the volume of the cells on level l.

The algorithm for boundaries is determined by the needs of AMRNodeElliptic. Right hand side values at boundary nodes are never used, so charge is not deposited to these nodes.

Since the charge density is a field value, projecting values from fine to coarse levels only requires copying the values on the shared nodes. The reverse direction should never be done, since the right hand side is always computed on the finest level available.

4.3.1 implementation details

As in mesh generation, this algorithm is straightforward in serial, but complex in parallel. The complexity stems from having to redistribute the particle data from its input processor distribution to match the processor distribution of the grids. This requires an interprocessor communication pattern that is all-to-some, which is difficult to implement efficiently when the "some" is small compared to "all", which is assumed to be the case here[6].

An alternate approach would be to have each processor deposit the charges for all the particles it owns, then distribute the results to the processors owning the grids. This could be done as a parallel reduction operation, but would still have to be done separately for each processor.

Since the underlying Boxes in AMRNodeElliptic are cell-centered, not node-centered, the distribution can be done by first making a BinFab of the particles and then iterating through the underlying Box and distributing the charge for each particle stored in the BinFab for that cell to all the nodes surrounding the cell.

This is done only for cells that are not covered by a finer level cell, and for "valid" nodes on coarse-fine boundaries. The function BinFab::AddItemsDestructive (or a variation thereof) can be used to construct the BinFabs for each level such that each particle is included in one BinFab.

For cells that contain embedded boundaries, the charge on the embedded nodes will be added to the nearest non-embedded node.

Note that finest-level nodes on coarse-fine interfaces that do not have a corresponding coarse node *are not* used in calculating the field. These are called "invalid" nodes. Although it is desirable to redistribute the charge on invalid nodes to nearby valid nodes, it is assumed to be a minor effect and may be ignored if it will take too much effort to implement.

To implement charge distribution in parallel requires transferring particle data to the processors containing the grids so the BinFabs can be generated on the processors that require them. Alternately, the BinFabs could be generated on the processors containing the particle data, then redistributed to the processors containing the grids, I think it is unlikely the latter would be more efficient.

An algorithm to distribute charge in parallel is:

```
loop over AMR levels
    foreach Box in the BoxLayout for this level, if the Box is on-processor, make a
    BinFab, else send the particles in this Box to the processor that owns the Box
    receive all particles and accumulate into BinFabs
    foreach BinFab on this processor, foreach particle in the BinFab, distribute charge
    to nodes
```

[6]because of the assumption that the particles are "mostly" sorted in physical space(see §2.2)

4.4 solve a Poission problem at the grid nodes to get the potential

AMRNodeElliptic solves $-\nabla^2\Phi = \rho$.

> Note: For now, we assume the mesh will not change during the Poisson solve. If it did, MeshRefine would have to be modified to incorporate the number of particles per cell into the tags used to refine/coarsen cells. This is essentially the same problem as in §4.2. In this case, the algorithm of §4.2 can be improved by storing the number or particles at all cells in a BaseFab<int>. In parallel, this allows the calculation of tag values to be done completely in-processor. Once a mesh hierarchy is generated that satisfies the maximum-particle criterion, it can be maintained during mesh refinement because refining a cell will never cause a violation[7], and tagging will ensure that coarsening a cell doesn't cause a violation.

4.5 compute $\vec{\mathbf{E}} = -\nabla\Phi$ at the grid nodes

AMRNodeElliptic has a Chombo Fortran function `GETGRAD` to compute a gradient accounting for the embedded boundary conditions. I assume it can be used here.

4.6 interpolate $\vec{\mathbf{E}}$ back to the particle locations

Interpolating the electric field $\vec{\mathbf{E}}$ from grid nodes to particle locations will be done using trilinear interpolation. For cells cut by embedded boundaries, it will be necessary to extrapolate from the interior nodes or get field values on the boundary from the boundary condition information. This will depend on whether the application can provide the BC info. The current `WARP` implementation only provides stencil coefficients for the discrete laplacian, but I assume it will not require a large effort to modify.

4.7 transfer $\vec{\mathbf{E}}$ at the paticle locations to the application

This requires extracting the values from the BinFabs and storing them in the output argument(s) in the order the application expects. The ChargedParticle class will save the information necessary to reorder the particle data.

The algorithm is:

- loop over AMR levels; loop over boxes in each level; loop over cells in each box
- extract List<ChargedParticle> for this cell and walk it
- foreach ChargedParticle, if the original processor is the current processor, save the $\vec{\mathbf{E}}$ data to the output argument according to the original index data, else save the ChargedParticle on a list for that original processor. [Note: use a Vector<List<ChargedParticle>>?]

[7]The number of particles in the refined cells can be computed by dividing the number in the coarse cell. This will be incorrect if the particles are not uniformly distributed in the cell, but it will not result in a violation of the maximum-particle criterion so it doesn't matter.

- foreach nonempty list, `send` the list to the appropriate processor
- concatenate all `receive`d lists
- foreach list element, save the $\vec{\mathbf{E}}$ data to the output argument according to the original index data

4.8 other issues

Both of the interpolation operations (4.3,4.6) will have to account for the embedded physical boundaries. `AMRNodeElliptic` already handles this, using functionality from `WARP` to determine which of the nearby grid nodes is outside an embedded boundary. The existing algorithms should be easy to reuse or adapt. The charge on a grid node that is outside a boundary will be added to the nearest interior node.

To enforce the maximum number of particles per cell will require modifying the algorithm that generates tags for the `MeshRefine` routine. At each mesh level, after the tags are generated from the residual, additional tags must be added on each level except the finest for every cell with too many particles. This ensures that a cell won't be coarsened if the coarsened cell would have too many particles.

5 Implementation for handling particles

5.1 data class for particles

We need a data class to represent a particle. This will inherit from BinItem in ParticleTools. This class must store the physical coordinates of the particle, the computed field variable Ev, an index or pointer to the particle's location in the application-level data structures, and a processor number. The last two values are needed so the Ev data can be copied back to the application's data structures after being computed. Precisely how this is done will depend on the application.

```
class ChargedParticle : public BinIntem { //inherits position
protected:
   RealVect m_efield ;
   int m_orig_index, m_orig_procnum ;
public:
   ChargedParticle();
   ChargedParticle(const RealVect& a_position, int a_index, int a_procnum );
   void define(const RealVect& a_position, int a_index, int a_procnum );
   void setEfield(const RealVect& a_efield);
   const RealVect& getEfield() const;
   int getOrigIndex() const ;
   int getOrigProcnum() const ;
}
```

5.2 BinFab modifications

To simplify interfacing to the Fortran applications, the BinFab class will be modified to add a version of addItems() that takes the particle positions as vector arguments instead of a generic List. To allow for the various Fortran interfaces, it should also allow for non-unit strides in the vector. This will be faster than copying all the data into a list just so it can be copied again into the BinFab, and implementation requires only minor modifications of the existing code.

5.3 Performance

If care is not taken, moving data from particles to grids can easily become the dominant cost of the whole method, even though it only happens once at the beginning and once at the end. The all-to-all message pattern that would result from the straightforward approach to this problem can be very expensive on large machines. A secondary concern is the size of the messages that get sent, but I think bandwidth is a lesser concern than latency.

6 Interface Specifications

In each of the following, the Fortran specification is first, followed by the C/C++ signature for the same routine. *I've taken liberties with the ChomboFortran syntax to improve readability.*

This is the primary interface between the Fortran applications and Chombo. It has several versions, corresponding to different data layouts. The first will be implemented initially.

```
subroutine CALCEFIELD0( N, XStride, X,Y,Z ,EStride ,ex,ey,ez ,status )
 integer N                                          !(input) number of particles
 integer XStride                                    !(input) memory stride for X,Y,Z
 REAL_T X(XStride,N), Y(XStride,N), Z(XStride,N)    !(input) particle coordinates
 integer EStride                                    !(input) memory stride for ex,ey,ez
 REAL_T ex(EStride,N), ey(EStride,N), ez(EStride,N) !(output) E field at particles
 integer status                                     !(output) result code

void CHF_FORTNAME(CALCEFIELD0)( const int& N, const int& XStride,
                                const REAL* X, Y, Z,
                                const int& EStride,
                                REAL* ex, ey, ez,
                                int& status ).
```

```
subroutine CALCEFIELD1( N, X,Y,Z ,ex,ey,ez ,status )
  integer N                    !(input) number of particles
  REAL_T X(N), Y(N), Z(N)      !(input) particle coordinates
  REAL_T ex(N), ey(N), ez(N)   !(output) E field at particles
  integer status               !(output) result code

void CHF_FORTNAME(CALCEFIELD1)( const int& N,
                                const REAL* X, Y, Z,
                                REAL* ex, ey, ez,
                                int& status ).
```

```
subroutine CALCEFIELD2( N, Stride, X ,e ,status )
  integer N               !(input) number of particles
  integer Stride          !(input) memory stride in X and e
  REAL_T X(Stride,N)      !(input) particle coordinates (x,y,z)
  REAL_T e(Stride,N)      !(output) E field at particles (x,y,z)
  integer status          !(output) result code

void CHF_FORTNAME(CALCEFIELD2)( const int& N, const int& Stride,
                                const REAL* X,
                                REAL* e,
                                int& status ).
```

The next function allows `Chombo` to ask the application for the number of particles per cell in a single grid.

```
subroutine GETPARTICLECOUNTS( Nc,XL,Dx,pc )
  INTEGER Nc(3)                     !(input) number of cells in each grid dimension
  REAL_T XL(3)                      !(input) coordinates of lower corner of grid
  REAL_T Dx(3)                      !(input) grid spacing
  INTEGER pc(Nc(1),Nc(2),Nc(3))     !(output) particles per cell

void CHF_FORTNAME(GETPARTICLECOUNTS)( const int* Nc,
                                      const REAL* XL, const REAL* Dx,
                                      int* pc)
```

Other interfaces are needed to pass constant parameters from the application to `Chombo` and allow `Chombo` to request interpolation coefficients.

```
subroutine SETCONSTANTS( Rho, MaxParticlesPerCell )
  REAL_T Rho,                 !(input) charge per particle (includes 4π factor)
  int MaxParticlesPerCell     !(input) max allowed by applicatoin

void CHF_FORTNAME(SETCONSTANTS)( const REAL& Rho,
                                 const int& MaxParticlesPerCell )
```

subroutine **SETDOMAIN**(XL,XH)
 REAL_T XL(CH_SPACEDIM), !(input) *coordinates of low corner*
 REAL_T XH(CH_SPACEDIM) !(input) *coordinates of high corner*

void CHF_FORTNAME(**SETDOMAIN**)(const REAL XL[CH_SPACEDIM],
 const REAL XH[CH_SPACEDIM])

This is the analog of `nodalcoefficients` for the trilinear interpolation.
subroutine **INTERPCOEFFICIENTS**(Dx, X, coeffs)
 REAL_T Dx, !(input) *mesh spacing*
 REAL_T X(CH_SPACEDIM) !(input) *coordinates of interpolation point*
 REAL_T coeffs(2**CH_SPACEDIM) !(output) *interpolation coeffients*

void CHF_FORTNAME(**INTERPCOEFFICIENTS**)(const REAL& Dx,
 const REAL X[CH_SPACEDIM],
 REAL& coeffs)

Note: coeffs are stored in column-major order.