

PARTICLE IN-CELL SIMULATION WITH LUCID

JOHN PLAICE

Département d'informatique, Université Laval

Ste-Foy (Québec), Canada G1K 7P4

E-mail: John.Plaice@ift.ulaval.ca

and

JOEY PAQUET

E-mail: jpaquet@ift.ulaval.ca

ABSTRACT

This paper presents a Lucid program for simulating plasmas using the particle in-cell technique. The program was derived from TRISTAN, a FORTRAN program that essentially runs the Maxwell–Hertz–Heaviside equations in finite difference form. The resulting Lucid program is 1/5 the size of the FORTRAN program, is much closer to the original differential equations and offers greater possibilities for exploiting parallelism.

1. Introduction

One of the original purposes for the Lucid language was to demonstrate that numerical problems could best be programmed and computed using the dataflow paradigm. However, most of the recent work on Lucid has dealt with the extension of the intensional programming paradigm, perhaps to the detriment of the original numeric problems.

In physics, however, the concept of dataflow is perfectly natural. Electromagnetic fields, gravitational fields and hydrodynamic flows are all examples of continuous flows that are simulated using assumptions that are quite familiar to people who understand dataflow.

TRISTAN is a proven FORTRAN program for particle in-cell simulation. Designed by the late Oscar Buneman of Stanford University, it has been used to simulate plasmas of different sizes, and is one of the first computer tools to offer a plausible explanation for galaxy formation.

However, the TRISTAN code is far more complex than are the Maxwell–Hertz–Heaviside equations upon which it is based. In fact, most of the FORTRAN code pertains to manipulating five-dimensional streams using assignments to FORTRAN arrays.

In the subsequent sections, an outline of how to translate TRISTAN into Lucid is presented. The two major results are that the Lucid code closely resembles the original differential equations and the level of parallelization increases from a fixed number of 27 to the number of cells, often more than a million, while increasing the level of locality.

2. Particle in-cell simulation

The basis for the TRISTAN code are the Maxwell–Hertz–Heaviside equations for electromagnetic fields:

$$\begin{aligned}\frac{\partial \mathbf{B}}{\partial t} &= -\nabla \times \mathbf{E}, \\ \frac{\partial \mathbf{D}}{\partial t} &= \nabla \times \mathbf{H} - \mathbf{j}, \\ \nabla \cdot \mathbf{D} &= \rho, \\ \nabla \cdot \mathbf{B} &= 0;\end{aligned}$$

and the Lorentz equations for motion:

$$\begin{aligned}\frac{d}{dt}(m\mathbf{v}) &= q(\mathbf{E} + \mathbf{v} \times \mathbf{B}), \\ \frac{d\mathbf{r}}{dt} &= \mathbf{v}.\end{aligned}$$

These differential equations are in a form that is well-suited to programming computers, and this was done by Oscar Buneman in a FORTRAN program called TRISTAN, using standard interpolation techniques for generating finite difference equations.

In the TRISTAN program, ¹ electricity (\mathbf{E}) (resp. magnetism (\mathbf{B})) is stored as three arrays \mathbf{ex} , \mathbf{ey} and \mathbf{ez} (resp. \mathbf{bx} , \mathbf{by} and \mathbf{bz}) that vary in the three dimensions x , y and z . At each time iteration, these arrays are re-computed.

To ensure that the discrete computations accurately reflect the continuous differential equations, the values in these cells are staggered as follows:

$$\begin{aligned}\mathbf{ex}(i, j, k) &\equiv \text{value of } \mathbf{ex} \text{ at } x = i + .5 & y = j & z = k \\ \mathbf{ey}(i, j, k) &\equiv \text{value of } \mathbf{ey} \text{ at } x = i & y = j + .5 & z = k \\ \mathbf{ez}(i, j, k) &\equiv \text{value of } \mathbf{ez} \text{ at } x = i & y = j & z = k + .5 \\ \mathbf{bx}(i, j, k) &\equiv \text{value of } \mathbf{bx} \text{ at } x = i & y = j + .5 & z = k + .5 \\ \mathbf{by}(i, j, k) &\equiv \text{value of } \mathbf{by} \text{ at } x = i + .5 & y = j & z = k + .5 \\ \mathbf{bz}(i, j, k) &\equiv \text{value of } \mathbf{bz} \text{ at } x = i + .5 & y = j + .5 & z = k\end{aligned}$$

which corresponds to the diagram in Figure 1. In these cells, charged particles, both ions and electrons, navigate. Their direction and velocity are affected by the surrounding electromagnetic forces, and their charge in turn affects those same forces. In the TRISTAN program, there are six arrays for the particles: (x, y, z) for their location (essentially an index into the field arrays) and (u, v, w) for their velocity.

Most of the known matter in the universe is in plasma form. Therefore, TRISTAN can be used to simulate phenomena at quite divergent scales, such as galaxy formation, van Allen belt radiation and fusion experiments. As far as simulation is concerned, it is the initial and boundary conditions that differ. Since we are presenting Lucid, we will suppose that people who better understand the physics can deal with these problems.

Once the initial conditions have been defined and checked to ensure that they meet the constraints expressed in the above equations, the program iterates for as many times as the user wishes. In the original FORTRAN program, each iteration consisted of:

1. half-advancing the magnetic field equations;
2. moving the particles;
3. half-advancing the magnetic field equations;
4. advancing the electric field equations;
5. adjusting the electric fields according to the particle motion.

The magnetic field half-advance takes place so that the electrical and magnetic forces are not just staggered in space, but also in time.

The fundamental difference between the FORTRAN and Lucid programs is that Lucid is intensional, so indices almost never need to be referred to.

Here is the core of the Lucid program:

```
B = B_init fby.dt lindman_curl(B, E, dNext, 0, MAX);
E = E_init fby.dt lindman_curl(E, next.dt B, dPrev, MAX, 0)
  - qdelta(X0traj, Xtraj, Qtraj);
```

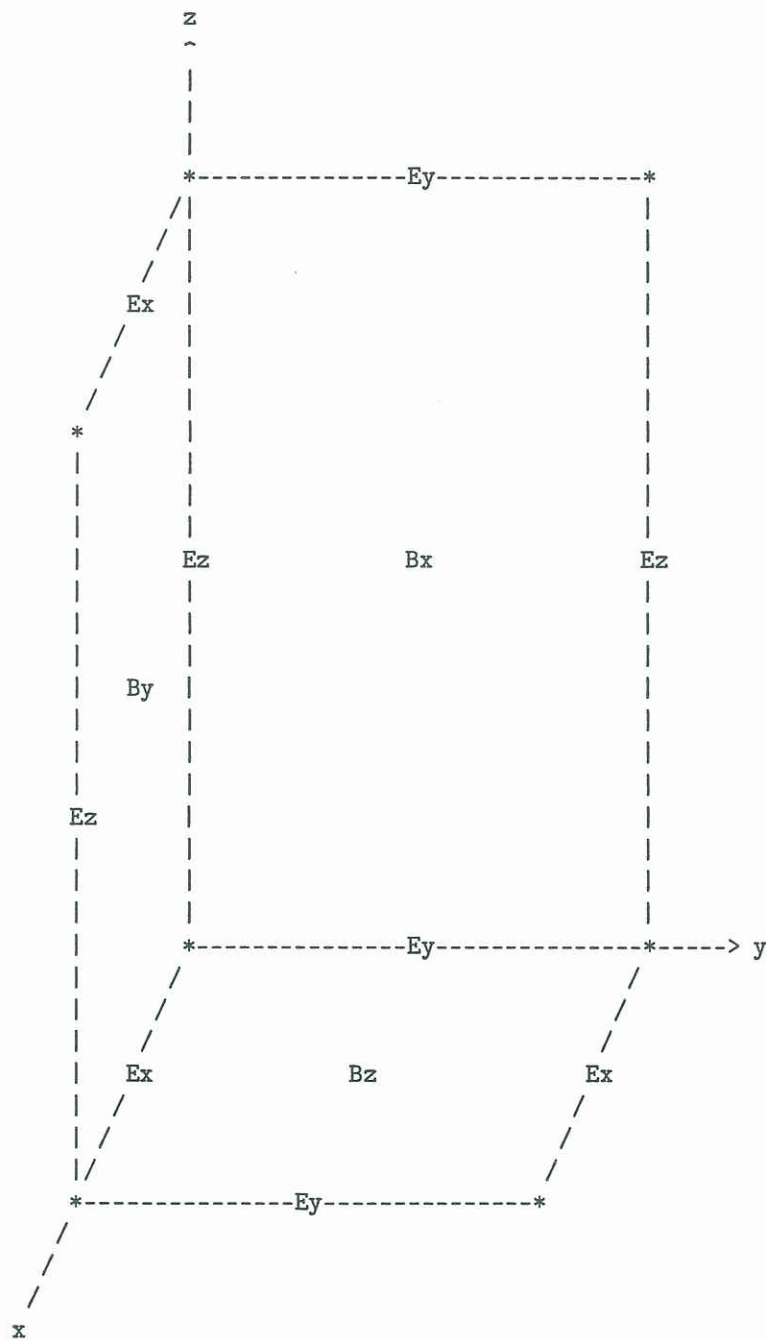


Figure 1: Staggering the entries for elctrical and magnetic fluxes.


```

{X,U,Q} = {X_init,U_init,Q_init} fby.dt
      doxyz(select,doxyz(combine,{Xnew,Unew,Q}));
{X0traj,Xtraj,Qtraj} = doxyz(split,{X,Xnew,Q});
{Xnew,Unew} = move(X, U, avgNext.dt B, E);

```

where B is the magnetic flux, E is the electrical flux, and X, U and Q are, respectively, the position, the velocity and the charge of the particles. All of these values are streams that vary in the dx, dy and dz space directions, the dt time direction and the dd component direction (3 possible values). The particle values also use a sixth dimension dl to designate the individual articles within a cell.

The core program will be examined in more detail below.

3. Multiple dimensions

As was mentioned above, most values in this program vary in five or six dimensions. To simplify the presentation that follows, the dimensions and the operations manipulating them are presented here.

First, the dimensions are designated.

```

dx = 0; i = #.dx;
dy = 1; j = #.dy;
dz = 2; k = #.dz;
dd = 3; space = #.dd;
dt = 4; time = #.dt;
dl = 5; list = #.dl;

```

Unlike all previous versions of Lucid, dimensions here have become first-class values. The 0-th dimension is 0, the 3-rd is 3, and so on. So dx is dimension 0 and i is the value of the index into dimension 0 at any given moment.

Dimensions dx, dy and dz correspond to the three space dimensions and will, in this program, vary between 0 and MAX. Dimension dd designates co-ordinates: it only uses values 0, 1 and 2, for, respectively, x, y and z. Dimension dt corresponds to time. Finally dl is used for manipulating streams of particles within individual cells.

Physics is full of situations where forces must be rotated. This is easily done with first-class dimensions. The values straight, right and left correspond to not rotating, rotating to the right or rotating to the left. Values M, R and L refer to the first three indexes in the (possibly) rotated object.

```

straight= space;
right = (#.dd+1) mod 3;
left = (#.dd+2) mod 3;
M = #.straight;
R = #.right;
L = #.left;
inv dir = (3-dir) mod 3;
rotRight E = component.right E;
rotLeft E = component.left E;

```

Here are a few basic operations that are used regularly.

```

doxyz(f,X) = f.dx f.dy f.dz X;
component.d E = E @.space d;
dNext.d E = next.d E - E;
dPrev.d E = E - prev.d E;
avgPrev.d E = 0.5 * (prev.d E + E);
avgNext.d E = 0.5 * (E + next.d E);

```

4. Field-updates

The simple differential equation

$$\frac{\partial \mathbf{B}}{\partial t} = -\nabla \times \mathbf{E}$$

becomes, in FORTRAN:

```
6  do 7 i=1,mx-1
    do 7 j=1,my-1
        do 7 k=1,mz-1
            bx(i,j,k)=bx(i,j,k) + (.5*c) *
&            (ey(i,j,k+1)-ey(i,j,k)-ez(i,j+1,k)+ez(i,j,k))
            by(i,j,k)=by(i,j,k) + (.5*c) *
&            (ez(i+1,j,k)-ez(i,j,k)-ex(i,j,k+1)+ex(i,j,k))
7  bz(i,j,k)=bz(i,j,k) + (.5*c) *
&            (ex(i,j+1,k)-ex(i,j,k)-ey(i+1,j,k)+ey(i,j,k))
```

In Lucid, these equations simply become

```
B - c * curl(dNext,E)
where
    curl(f,E) =
        diff(f,left,E) - diff(f,right,E);
    diff(f,dir,E) =
        f(inv dir, component.dir E);
end;
```

The curl of E is given by

$$\text{curl } \mathbf{F} = \hat{x} \left(\frac{\partial E_z}{\partial y} - \frac{\partial E_y}{\partial z} \right) + \hat{y} \left(\frac{\partial E_x}{\partial z} - \frac{\partial E_z}{\partial x} \right) + \hat{z} \left(\frac{\partial E_y}{\partial x} - \frac{\partial E_x}{\partial y} \right).$$

When the differentials are computed using finite difference equations, then the values must be staggered, as in Figure 1. Because of the offset, the equations for computing the curl of E and B are not quite identical: the first uses `dPrev` and the second uses `dNext`.

The differential equations apply throughout infinite space. However, simulations require a bounded space. To ensure that bounded space does not create weird situations, special equations must apply at or near the limits. The equations used here are from Lindman's boundary methods (function `lindman_surface`).

```
lindman_curl(B, E, dfun, min, max) =
    if L ne max and R ne max
    then B - c * curl(dfun,E)
    elif not minormax(M) and
        ((L eq max and not minormax(R)) or
         (R eq max and not minormax(L)))
    then lindman_surface(B,E)
    else edge next.dt B
    fi
where
    curl(f,E) =
        diff(f,left,E) - diff(f,right,E);
    diff(f,dir,E) =
        f(inv dir, component.dir E);
```

```

lindman_surface(B,E) =
    prev.up next.dt B
    + rs * (B - prev.up next.dt B)
    + ps * (avgNext.dt dPrev.straight rotate.up B)
    + lsign * os * (dNext.other rotate.up E)
    + lsign * (os-c) * (prev.up dNext.other rotate.up E)
    + lsign * c * (dNext.up rotate.other E);
{other,up,lsign} =
    if L eq max
    then {right,left,-1.0}
    else {left,right,1.0}
    fi;
edge E =
    if minormax(M)
    then avg_edge(M,L,E)
    else avg_edge(L,R,E)
    fi;
avg_edge(dir1,dir2,E) =
    f.dir1 E + f.dir2 E - f.dir2 f.dir2 E
    where
        f.d = if d eq min then next.d else prev.d fi;
    end;
minormax(n) =
    n eq min or n eq max;
end;

```

5. Moving particles

The actual moving of the particles is a very simple process. However, before the particles can be moved, the system must compute the exact electromagnetic forces for each particle. This is done through an interpolation process that is made more complex by the fact that the electrical and magnetic fields are staggered.

The field interpolations are tri-linear (linear in x , linear in y and linear in z), where linear in one direction becomes:

$$f_{i+\delta} = f_i + \delta(f_{i+1} - f_i).$$

However, when the field is staggered, the result becomes:

$$f_{i+\delta} = \frac{f_i + f_{i-1} + \delta(f_{i+1} - f_{i-1})}{2}.$$

So, the FORTRAN code

```

C   E-component interpolations:
f=ex(1)+ex(1-ix)+dx*(ex(1+ix)-ex(1-ix))
f=f+dy*(ex(1+iy)+ex(1-ix+iy)+dx*(ex(1+ix+iy)-ex(1-ix+iy))-f)
g=ex(1+iz)+ex(1-ix+iz)+dx*(ex(1+ix+iz)-ex(1-ix+iz))
g=g+dy*
& (ex(1+iy+iz)+ex(1-ix+iy+iz)+dx*(ex(1+ix+iy+iz)-ex(1-ix+iy+iz))-g)
ex0=(f+dz*(g-f))*(.25*qm)

```



```

C -----
  f=ey(1)+ey(1-iy)+dy*(ey(1+iy)-ey(1-iy))
  f=f+dz*(ey(1+iz)+ey(1-iy+iz)+dy*(ey(1+iy+iz)-ey(1-iy+iz))-f)
  g=ey(1+ix)+ey(1-iy+ix)+dy*(ey(1+iy+ix)-ey(1-iy+ix))
  g=g+dz*
  & (ey(1+iz+ix)+ey(1-iy+iz+ix)+dy*(ey(1+iy+iz+ix)-ey(1-iy+iz+ix))-g)
  ey0=(f+dx*(g-f))*(.25*qm)
C -----
  f=ez(1)+ez(1-iz)+dz*(ez(1+iz)-ez(1-iz))
  f=f+dx*(ez(1+ix)+ez(1-iz+ix)+dz*(ez(1+iz+ix)-ez(1-iz+ix))-f)
  g=ez(1+iy)+ez(1-iz+iy)+dz*(ez(1+iz+iy)-ez(1-iz+iy))
  g=g+dx*
  & (ez(1+ix+iy)+ez(1-iz+ix+iy)+dz*(ez(1+iz+ix+iy)-ez(1-iz+ix+iy))-g)
  ez0=(f+dy*(g-f))*(.25*qm)
and
C B-component interpolations:
  f=bx(1-iy)+bx(1-iy-iz)+dz*(bx(1-iy+iz)-bx(1-iy-iz))
  f=bx(1)+bx(1-iz)+dz*(bx(1+iz)-bx(1-iz))+f+dy*
  & (bx(1+iy)+bx(1+iy-iz)+dz*(bx(1+iy+iz)-bx(1+iy-iz))-f)
  g=bx(1+ix-iy)+bx(1+ix-iy-iz)+dz*(bx(1+ix-iy+iz)-bx(1+ix-iy-iz))
  g=bx(1+ix)+bx(1+ix-iz)+dz*(bx(1+ix+iz)-bx(1+ix-iz))+g+dy*
  & (bx(1+ix+iy)+bx(1+ix+iy-iz)+dz*(bx(1+ix+iy+iz)-bx(1+ix+iy-iz))-g)
  bx0=(f+dx*(g-f))*(.125*qm/c)
C -----
  f=by(1-iz)+by(1-iz-ix)+dx*(by(1-iz+ix)-by(1-iz-ix))
  f=by(1)+by(1-ix)+dx*(by(1+ix)-by(1-ix))+f+dz*
  & (by(1+iz)+by(1+iz-ix)+dx*(by(1+iz+ix)-by(1+iz-ix))-f)
  g=by(1+iy-iz)+by(1+iy-iz-ix)+dx*(by(1+iy-iz+ix)-by(1+iy-iz-ix))
  g=by(1+iy)+by(1+iy-ix)+dx*(by(1+iy+ix)-by(1+iy-ix))+g+dz*
  & (by(1+iy+iz)+by(1+iy+iz-ix)+dx*(by(1+iy+iz+ix)-by(1+iy+iz-ix))-g)
  by0=(f+dy*(g-f))*(.125*qm/c)
C -----
  f=bz(1-ix)+bz(1-ix-iy)+dy*(bz(1-ix+iy)-bz(1-ix-iy))
  f=bz(1)+bz(1-iy)+dy*(bz(1+iy)-bz(1-iy))+f+dx*
  & (bz(1+ix)+bz(1+ix-iy)+dy*(bz(1+ix+iy)-bz(1+ix-iy))-f)
  g=bz(1+iz-ix)+bz(1+iz-ix-iy)+dy*(bz(1+iz-ix+iy)-bz(1+iz-ix-iy))
  g=bz(1+iz)+bz(1+iz-iy)+dy*(bz(1+iz+iy)-bz(1+iz-iy))+g+dx*
  & (bz(1+iz+ix)+bz(1+iz+ix-iy)+dy*(bz(1+iz+ix+iy)-bz(1+iz+ix-iy))-g)
  bz0=(f+dz*(g-f))*(.125*qm/c)

```

simply becomes

```

B_real = avgPrev.left avgPrev.right B;
E_real = avgPrev.straight E;
B0 = force(B_real,X) * Q / c;
E0 = force(E_real,X) * Q;
force(E,location) = E2
where
  E2 = doxyz(interpolate,E1);
  E1 = doxyz(shift,E);
  delta = frac location;
  cell = int location;
  shift.d E = E @.d component.d cell;
  interpolate.d E = E + component.d delta * dNext.d E;
end;

```

B_real and E_real are the ‘de-staggered’ values for B and E.

Once the values of the magnetic and electrical fluxes are computed, then the particles must be moved. The sequence of derivations is as follows:

$$\begin{aligned}
\gamma_1 &= \frac{1}{\sqrt{1 - \frac{v^2}{c^2}}} \\
\mathbf{u}_1 &= \gamma_1 \mathbf{v}_{\text{old}} \\
\mathbf{u}_2 &= \mathbf{u}_1 + \mathbf{E} \\
\gamma_2 &= \sqrt{1 + \frac{u_2^2}{c^2}} \\
\mathbf{u}_3 &= \left(\frac{2}{\gamma_2^2 + B^2} \right) (\gamma_2 \mathbf{u}_2 + \mathbf{u}_2 \times \mathbf{B}) \times \mathbf{B} \\
\mathbf{u}_4 &= \mathbf{u}_3 + \mathbf{E} \\
\mathbf{v}_{\text{new}} &= \mathbf{u}_4 \sqrt{1 + \frac{u_4^2}{c^2}} \\
\mathbf{x}_{\text{new}} &= \mathbf{x}_{\text{old}} + \delta t \mathbf{v}_{\text{new}}
\end{aligned}$$

Here is the result in Lucid:

```

move(X,U,B,E) = {Xnew,Unew}
where
  B_real = avgPrev.left avgPrev.right B;
  E_real = avgPrev.straight E;
  B0 = force(B_real,X);
  E0 = force(E_real,X);
  g0 = c / sqrt(c**2 - norm U);
  U0 = g0*U + E0;
  g1 = c / sqrt(c**2 + norm U0);
  B1 = g1*B0;
  f = 2.0 / (1.0 + norm B1);
  U1 = f * (U0 + cross(U0,B1));
  U2 = E0 + U0 + cross(U1,B1);
  g2 = c / sqrt(c**2 + norm U2);
  U3 = g2*U2;
  X3 = X + U3;
  Unew = U3 * sign(1.0,X3-3.0) * sign(1.0,MAX-X3-2.0);
  Xnew = MAX-2.0-abs(MAX-5.0-abs(X3-3.0));

```



```

abs(E) = if E ge 0.0 then E else -E fi;
sign(x,E) = if E ge 0.0 then abs(x) else -abs(x) fi;
force(E,location) = E2
where
  E2 = doxyz(interpolate,E1);
  E1 = doxyz(shift,E);
  delta = frac location;
  cell = int location;
  shift.d E = E @.d component.d cell;
  interpolate.d E = E + component.d delta * dNext.d E;
end;
norm E = square(Ex) + square(Ey) + square(Ez)
where
  Ex=component.dx E;
  Ey=component.dy E;
  Ez=component.dz E;
end;
cross(E1,E2) = (rotRight E1)*(rotLeft E2)
               - (rotLeft E1)*(rotRight E2);
end;

```

6. Charge updates

The TRISTAN code modifies the electrical field by passing through the particles, one at a time, and calculating the effects of that particle on the surrounding cells. The equations are set up in such a way that a particle cannot advance more than half a cell-width in one iteration, therefore no particle may travel to a cell that is not adjacent.

Each particle will move during an iteration. Its arrival point may be an adjacent cell, and to get there, it will possibly pass through other (adjacent) cells. For each affected cell, i.e. where a particle is tourist or immigrant, the charge of that particle affects the electrical field of that cell, as well as that of its 26 immediate neighbors. These computations can be run in parallel, thereby yielding the 27-fold parallelism in TRISTAN.

However, in Lucid, the basis for parallelism is the cell. Each cell has a list of the particles that are contained in it, and each cell inquires from its neighbors what are the particles that affect it. Here is the main program again:

```

B = B_init fby.dt lindman_curl(B, E, dNext, 0, MAX);
E = E_init fby.dt lindman_curl(E, next.dt B, dPrev, MAX, 0)
    - qdelta(X0traj, Xtraj, Qtraj);
{X,U,Q} = {X_init,U_init,Q_init} fby.dt
    doxyz(select,dxyz(combine,{Xnew,Unew,Q}));
{X0traj,Xtraj,Qtraj} = doxyz(split,{X,Xnew,Q});
{Xnew,Unew} = move(X, U, avgNext.dt B, E);

```

For a given instant and cell, the X0traj and Xtraj values correspond to the beginning and the end of the trajectories of all the particles that visit that cell in that instant. Once these values are known, it is possible to compute the actual effect on the electrical field:

```

qdelta(X0traj,Xtraj,Qtraj) = total_delta
where
  diffX = Xtraj - X0traj;
  I = 0.5 * diffX;
  dX = 0.5 * (Xtraj + X0traj) - I;
  dY = rotRight dX;
  dZ = rotLeft dX;
  qU = Qtraj * diffX;
  delta = 0.08333333 * Qtraj
    * component.dx diffX
    * component.dy diffX
    * component.dz diffX;
  delta1 = compute_delta(dY, dZ, delta);
  delta2 = compute_delta(1.0-dY, dZ, -delta);
  delta3 = compute_delta(dY, 1.0-dZ, -delta);
  delta4 = compute_delta(1.0-dY, 1.0-dZ, delta);
  total_delta = delta4
    + prev.right delta3
    + prev.left delta2
    + prev.right prev.left delta1;
  compute_delta(y,z,delta) =
    doxyz(smooth,sum.dl(qU * y*z + delta);
  smooth.d X = 0.25 * prev.d X
    + 0.5 * X
    + 0.25 * next.d X;
  sum.d X = Y asa.d iseod(X)
  where
    Y = 0.0 fby.d X + Y;
  end;
end;

```

For each cell, for each component (x, y or z), a particle passing through that cell will affect the electrical field values on all the surfaces of the cell. Therefore four values must be computed for each component. Hence values delta1 through delta4. The effects of the electrical field are then smoothed in the three space directions to avoid harmonics.

This method computes the effects of all the particles on a cell before updating the fields. Doing so significantly reduces the number of computations to make. Similarly, localizing all work to cells *massively* increases the level of parallelism (from 27 to — potentially — several million).

7. Particle selection

Finally, here are the remaining routines, used to make sure that at each iteration, each cell contains the correct particles.

```

combine.d X =
  merge.dl(prev.d X, merge.dl(X, next.d X));
select.d {X,U,Q} =
  {X,U,Q} wvr.d int(component.d X) eq #.d;

```

```

split.d {X0,X,Q} =
  if iseod X0
  then eod
  elif int(component.d X0) eq int(component.d X)
  then {X0,X,Q} fby.dl rest
  else {X0,X1,Q} fby.dl {X1,X,Q} fby.dl rest
  fi
where
  rest = split.d next.dl {X0,X,Q};
  X1 = if (space eq d)
    then 0.5 * real(1 + int(X0) + int(X))
    else X0 + (X-X0) * (x1-x0)/(x-x0)
    fi;
  x1 = component.d X1;
  x0 = component.d X0;
  x  = component.d X;
end;

```

8. Implementation

The above comments show that it is possible to take an existing FORTRAN program, and study and analyze it in order to produce a Lucid program that is significantly shorter and clearer, in which the level of parallelism is orders of magnitude greater. The full paper will provide more details about programming TRISTAN in Lucid.

However, there are no efficient fine-grain parallel implementations of Lucid. So a new problem has been introduced: how can one implement this program efficiently? There are two ways to approach this problem, each of which offers part of the solution.

First, the arrays that are being manipulated in this program are not general infinite structures but, rather, finite 5-dimensional structures whose bounds are known. Furthermore, the operations that are being used are all simple and, most importantly, local in action. Except for handling boundary conditions, no operation requires knowledge of any data but that of provided by neighboring cells. This means that a sequential C or FORTRAN program could be generated directly from the Lucid program.

Second, there is already available a coarse-grain implementation of Lucid: GLU. Using either a peer architecture or a manager-worker architecture, GLU can use education to have tasks written in C or FORTRAN run on parallel systems.

The solution then becomes simple. The Lucid TRISTAN program is compiled into a C program. The latter, upon reception of data, decides if the problem is too large for it to handle by itself. If it is too large, it breaks up the data into smaller pieces and calls GLU, which then farms off the work to other machines. As a result, a hierarchy of networks running in parallel can be set up, and communication overhead can be kept to a minimum. An attempt will also be made to distribute data physically, so that no data is ever concentrated on a single disk.

The full paper will catalogue the experimental results from this program. Furthermore, an attempt will be made to obtain real data so that real simulations can be made with Lucid.

9. References

1. Anthony L. Peratt. *Physics of the Plasma Universe*. Springer-Verlag, 1992.