```
/*--------------------------------------------------------------------*/
void cgrbppushf23l(float ppart[], float fxy[], float bxy[], int kpic[],
                   int ncl[], int ihole[], float qbm, float dt,
                   float dtc, float ci, float *ek, int idimp, int nppmx,
                   int nx, int ny, int mx, int my, int nxv, int nyv,
                   int mx1, int mxy1, int ntmax, int *irc) {
/* for 2-1/2d code, this subroutine updates particle co-ordinates and
   velocities using leap-frog scheme in time and first-order linear
   interpolation in space, for relativistic particles with magnetic field
   with periodic boundary conditions.
   Using the Boris Mover.
   also determines list of particles which are leaving this tile
   OpenMP version using guard cells
   data deposited in tiles
   particles stored segmented array
   131 flops/particle, 4 divides, 2 sqrts, 25 loads, 5 stores
   input: all except ncl, ihole, irc, output: ppart, ncl, ihole, irc, ek
   momentum equations used are:
   px(t+dt/2) = rot(1)*(px(t-dt/2) + .5*(q/m)*fx(x(t),y(t))*dt) +
      rot(2)*(py(t-dt/2) + .5*(q/m)*fy(x(t),y(t))*dt) +
      rot(3)*(pz(t-dt/2) + .5*(q/m)*fz(x(t),y(t))*dt) +
      .5*(q/m)*fx(x(t),y(t))*dt)
   py(t+dt/2) = rot(4)*(px(t-dt/2) + .5*(q/m)*fx(x(t),y(t))*dt) +
      rot(5)*(py(t-dt/2) + .5*(q/m)*fy(x(t),y(t))*dt) +
      rot(6)*(pz(t-dt/2) + .5*(q/m)*fz(x(t),y(t))*dt) +
      .5*(q/m)*fy(x(t),y(t))*dt)
   pz(t+dt/2) = rot(7)*(px(t-dt/2) + .5*(q/m)*fx(x(t),y(t))*dt) +
      rot(8)*(py(t-dt/2) + .5*(q/m)*fy(x(t),y(t))*dt) +
      rot(9)*(pz(t-dt/2) + .5*(q/m)*fz(x(t),y(t))*dt) +
      .5*(q/m)*fz(x(t),y(t))*dt)
   where q/m is charge/mass, and the rotation matrix is given by:
      rot[0] = (1 - (om*dt/2)**2 + 2*(omx*dt/2)**2)/(1 + (om*dt/2)**2)
      rot[1] = 2*(omz*dt/2 + (omx*dt/2)*(omy*dt/2))/(1 + (om*dt/2)**2)
      rot[2] = 2*(-omy*dt/2 + (omx*dt/2)*(omz*dt/2))/(1 + (om*dt/2)**2)
      rot[3] = 2*(-omz*dt/2 + (omx*dt/2)*(omy*dt/2))/(1 + (om*dt/2)**2)
      rot[4] = (1 - (om*dt/2)**2 + 2*(omy*dt/2)**2)/(1 + (om*dt/2)**2)
      rot[5] = 2*(omx*dt/2 + (omy*dt/2)*(omz*dt/2))/(1 + (om*dt/2)**2)
      rot[6] = 2*(omy*dt/2 + (omx*dt/2)*(omz*dt/2))/(1 + (om*dt/2)**2)
      rot[7] = 2*(-omx*dt/2 + (omy*dt/2)*(omz*dt/2))/(1 + (om*dt/2)**2)
      rot[8] = (1 - (om*dt/2)**2 + 2*(omz*dt/2)**2)/(1 + (om*dt/2)**2)
   and om**2 = omx**2 + omy**2 + omz**2
   the rotation matrix is determined by:
   omx = (q/m)*bx(x(t),y(t))*gami, omy = (q/m)*by(x(t),y(t))*gami, and
   omz = (q/m)*bz(x(t),y(t))*gami,
   where gami = 1./sqrt(1.+(px(t)*px(t)+py(t)*py(t)+pz(t)*pz(t))*ci*ci)
   position equations used are:
   x(t+dt) = x(t) + px(t+dt/2)*dtg
   y(t+dt) = y(t) + py(t+dt/2)*dtg
   where dtg = dtc/sqrt(1.+(px(t+dt/2)*px(t+dt/2)+py(t+dt/2)*py(t+dt/2)+
   pz(t+dt/2)*pz(t+dt/2))*ci*ci)
   fx(x(t),y(t)), fy(x(t),y(t)), and fz(x(t),y(t))
   bx(x(t),y(t)), by(x(t),y(t)), and bz(x(t),y(t))
   are approximated by interpolation from the nearest grid points:
   fx(x,y) = (1-dy)*((1-dx)*fx(n,m)+dx*fx(n+1,m)) + dy*((1-dx)*fx(n,m+1)
```

```
      + dx*fx(n+1,m+1))
   where n,m = leftmost grid points and dx = x-n, dy = y-m
   similarly for fy(x,y), fz(x,y), bx(x,y), by(x,y), bz(x,y)
   ppart[m][n][0] = position x of particle n in tile m
   ppart[m][n][1] = position y of particle n in tile m
   ppart[m][n][2] = x momentum of particle n in tile m
   ppart[m][n][3] = y momentum of particle n in tile m
   ppart[m][n][4] = z momentum of particle n in tile m
   fxy[k][j][0] = x component of force/charge at grid (j,k)
   fxy[k][j][1] = y component of force/charge at grid (j,k)
   fxy[k][j][2] = z component of force/charge at grid (j,k)
   that is, convolution of electric field over particle shape
   bxy[k][j][0] = x component of magnetic field at grid (j,k)
   bxy[k][j][1] = y component of magnetic field at grid (j,k)
   bxy[k][j][2] = z component of magnetic field at grid (j,k)
   that is, the convolution of magnetic field over particle shape
   kpic[k] = number of particles in tile k
   ncl[k][i] = number of particles going to destination i, tile k
   ihole[k][:][0] = location of hole in array left by departing particle
   ihole[k][:][1] = destination of particle leaving hole
   ihole[k][0][0] = ih, number of holes left (error, if negative)
   qbm = particle charge/mass ratio
   dt = time interval between successive calculations
   dtc = time interval between successive co-ordinate calculations
   ci = reciprical of velocity of light
   kinetic energy/mass at time t is also calculated, using
   ek = gami*sum((px(t-dt/2) + .5*(q/m)*fx(x(t),y(t))*dt)**2 +
        (py(t-dt/2) + .5*(q/m)*fy(x(t),y(t))*dt)**2 +
        (pz(t-dt/2) + .5*(q/m)*fz(x(t),y(t))*dt)**2)/(1. + gami)
   idimp = size of phase space = 5
   nppmx = maximum number of particles in tile
   nx/ny = system length in x/y direction
   mx/my = number of grids in sorting cell in x/y
   nxv = first dimension of field arrays, must be >= nx+1
   nyv = second dimension of field arrays, must be >= ny+1
   mx1 = (system length in x direction - 1)/mx + 1
   mxy1 = mx1*my1, where my1 = (system length in y direction - 1)/my + 1
   ntmax = size of hole array for particles leaving tiles
   irc = maximum overflow, returned only if error occurs, when irc > 0
   optimized version
local data                                                            */
#define MXV             33
#define MYV             33
   int noff, moff, npoff, npp, mxv3;
   int i, j, k, ih, nh, nn, mm, nm;
   float qtmh, ci2, dxp, dyp, amx, amy, dx, dy, dz, ox, oy, oz;
   float acx, acy, acz, p2, gami, qtmg, dtg, omxt, omyt, omzt, omt;
   float anorm, rot1, rot2, rot3, rot4, rot5, rot6, rot7, rot8, rot9;
   float anx, any, edgelx, edgely, edgerx, edgery;
   float x, y;
   float sfxy[3*MXV*MYV], sbxy[3*MXV*MYV];
/* float sfxy[3*(mx+1)*(my+1)], sbxy[3*(mx+1)*(my+1)]; */
   double sum1, sum2;
   mxv3 = 3*(mx + 1);
```

```
   qtmh = 0.5*qbm*dt;
   ci2 = ci*ci;
   anx = (float) nx;
   any = (float) ny;
   sum2 = 0.0;
/* error if local array is too small */
/* if ((mx >= MXV) || (my >= MYV)) */
/*    return;                       */
/* loop over tiles */
#pragma omp parallel for \
private(i,j,k,noff,moff,npp,npoff,nn,mm,nm,ih,nh,x,y,dxp,dyp,amx,amy, \
dx,dy,dz,ox,oy,oz,acx,acy,acz,omxt,omyt,omzt,omt,anorm,rot1,rot2,rot3, \
rot4,rot5,rot6,rot7,rot8,rot9,edgelx,edgely,edgerx,edgery,p2,gami, \
qtmg,dtg,sum1,sfxy,sbxy) \
reduction(+:sum2)
   for (k = 0; k < mxy1; k++) {
      noff = k/mx1;
      moff = my*noff;
      noff = mx*(k - mx1*noff);
      npp = kpic[k];
      npoff = nppmx*k;
      nn = nx - noff;
      nn = mx < nn ? mx : nn;
      mm = ny - moff;
      mm = my < mm ? my : mm;
      edgelx = noff;
      edgerx = noff + nn;
      edgely = moff;
      edgery = moff + mm;
      ih = 0;
      nh = 0;
      nn += 1;
      mm += 1;
/* load local fields from global array */
      for (j = 0; j < mm; j++) {
         for (i = 0; i < nn; i++) {
            sfxy[3*i+mxv3*j] = fxy[3*(i+noff+nxv*(j+moff))];
            sfxy[1+3*i+mxv3*j] = fxy[1+3*(i+noff+nxv*(j+moff))];
            sfxy[2+3*i+mxv3*j] = fxy[2+3*(i+noff+nxv*(j+moff))];
         }
      }
      for (j = 0; j < mm; j++) {
         for (i = 0; i < nn; i++) {
            sbxy[3*i+mxv3*j] = bxy[3*(i+noff+nxv*(j+moff))];
            sbxy[1+3*i+mxv3*j] = bxy[1+3*(i+noff+nxv*(j+moff))];
            sbxy[2+3*i+mxv3*j] = bxy[2+3*(i+noff+nxv*(j+moff))];
         }
      }
/* clear counters */
      for (j = 0; j < 8; j++) {
         ncl[j+8*k] = 0;
      }
      sum1 = 0.0;
/* loop over particles in tile */
```

```
      for (j = 0; j < npp; j++) {
/* find interpolation weights */
         x = ppart[idimp*(j+npoff)];
         y = ppart[1+idimp*(j+npoff)];
         nn = x;
         mm = y;
         dxp = x - (float) nn;
         dyp = y - (float) mm;
         nm = 3*(nn - noff) + mxv3*(mm - moff);
         amx = 1.0 - dxp;
         amy = 1.0 - dyp;
/* find electric field */
         nn = nm;
         dx = amx*sfxy[nn];
         dy = amx*sfxy[nn+1];
         dz = amx*sfxy[nn+2];
         mm = nn + 3;
         dx = amy*(dxp*sfxy[mm] + dx);
         dy = amy*(dxp*sfxy[mm+1] + dy);
         dz = amy*(dxp*sfxy[mm+2] + dz);
         nn += mxv3;
         acx = amx*sfxy[nn];
         acy = amx*sfxy[nn+1];
         acz = amx*sfxy[nn+2];
         mm = nn + 3;
         dx += dyp*(dxp*sfxy[mm] + acx);
         dy += dyp*(dxp*sfxy[mm+1] + acy);
         dz += dyp*(dxp*sfxy[mm+2] + acz);
/* find magnetic field */
         nn = nm;
         ox = amx*sbxy[nn];
         oy = amx*sbxy[nn+1];
         oz = amx*sbxy[nn+2];
         mm = nn + 3;
         ox = amy*(dxp*sbxy[mm] + ox);
         oy = amy*(dxp*sbxy[mm+1] + oy);
         oz = amy*(dxp*sbxy[mm+2] + oz);
         nn += mxv3;
         acx = amx*sbxy[nn];
         acy = amx*sbxy[nn+1];
         acz = amx*sbxy[nn+2];
         mm = nn + 3;
         ox += dyp*(dxp*sbxy[mm] + acx);
         oy += dyp*(dxp*sbxy[mm+1] + acy);
         oz += dyp*(dxp*sbxy[mm+2] + acz);
/* calculate half impulse */
         dx *= qtmh;
         dy *= qtmh;
         dz *= qtmh;
/* half acceleration */
         acx = ppart[2+idimp*(j+npoff)] + dx;
         acy = ppart[3+idimp*(j+npoff)] + dy;
         acz = ppart[4+idimp*(j+npoff)] + dz;
/* find inverse gamma */
```

```
         p2 = acx*acx + acy*acy + acz*acz;
         gami = 1.0/sqrtf(1.0 + p2*ci2);
/* renormalize magnetic field */
         qtmg = qtmh*gami;
/* time-centered kinetic energy */
         sum1 += gami*p2/(1.0 + gami);
/* calculate cyclotron frequency */
         omxt = qtmg*ox;
         omyt = qtmg*oy;
         omzt = qtmg*oz;
/* calculate rotation matrix */
         omt = omxt*omxt + omyt*omyt + omzt*omzt;
         anorm = 2.0/(1.0 + omt);
         omt = 0.5*(1.0 - omt);
         rot4 = omxt*omyt;
         rot7 = omxt*omzt;
         rot8 = omyt*omzt;
         rot1 = omt + omxt*omxt;
         rot5 = omt + omyt*omyt;
         rot9 = omt + omzt*omzt;
         rot2 = omzt + rot4;
         rot4 -= omzt;
         rot3 = -omyt + rot7;
         rot7 += omyt;
         rot6 = omxt + rot8;
         rot8 -= omxt;
/* new momentum */
         dx += (rot1*acx + rot2*acy + rot3*acz)*anorm;
         dy += (rot4*acx + rot5*acy + rot6*acz)*anorm;
         dz += (rot7*acx + rot8*acy + rot9*acz)*anorm;
         ppart[2+idimp*(j+npoff)] = dx;
         ppart[3+idimp*(j+npoff)] = dy;
         ppart[4+idimp*(j+npoff)] = dz;
/* update inverse gamma */
         p2 = dx*dx + dy*dy + dz*dz;
         dtg = dtc/sqrtf(1.0 + p2*ci2);
/* new position */
         dx = x + dx*dtg;
         dy = y + dy*dtg;
/* find particles going out of bounds */
         mm = 0;
/* count how many particles are going in each direction in ncl   */
/* save their address and destination in ihole                  */
/* use periodic boundary conditions and check for roundoff error */
/* mm = direction particle is going                              */
         if (dx >= edgerx) {
            if (dx >= anx)
               dx -= anx;
            mm = 2;
         }
         else if (dx < edgelx) {
            if (dx < 0.0f) {
               dx += anx;
               if (dx < anx)
```

```
                  mm = 1;
               else
                  dx = 0.0;
            }
            else {
               mm = 1;
            }
         }
         if (dy >= edgery) {
            if (dy >= any)
               dy -= any;
            mm += 6;
         }
         else if (dy < edgely) {
            if (dy < 0.0) {
               dy += any;
               if (dy < any)
                  mm += 3;
               else
                  dy = 0.0;
            }
            else {
               mm += 3;
            }
         }
/* set new position */
         ppart[idimp*(j+npoff)] = dx;
         ppart[1+idimp*(j+npoff)] = dy;
/* increment counters */
         if (mm > 0) {
            ncl[mm+8*k-1] += 1;
            ih += 1;
            if (ih <= ntmax) {
               ihole[2*(ih+(ntmax+1)*k)] = j + 1;
               ihole[1+2*(ih+(ntmax+1)*k)] = mm;
            }
            else {
               nh = 1;
            }
         }
         sum2 += sum1;
      }
/* set error and end of file flag */
/* ihole overflow */
      if (nh > 0) {
         *irc = ih;
         ih = -ih;
      }
      ihole[2*(ntmax+1)*k] = ih;
   }
/* normalize kinetic energy */
   *ek += sum2;
   return;
#undef MXV
```

```
#undef MYV
}
/*--------------------------------------------------------------------*/
void cgppost2l(float ppart[], float q[], int kpic[], float qm,
               int nppmx, int idimp, int mx, int my, int nxv, int nyv,
               int mx1, int mxy1) {
/* for 2d code, this subroutine calculates particle charge density
   using first-order linear interpolation, periodic boundaries
   OpenMP version using guard cells
   data deposited in tiles
   particles stored segmented array
   17 flops/particle, 6 loads, 4 stores
   input: all, output: q
   charge density is approximated by values at the nearest grid points
   q(n,m)=qm*(1.-dx)*(1.-dy)
   q(n+1,m)=qm*dx*(1.-dy)
   q(n,m+1)=qm*(1.-dx)*dy
   q(n+1,m+1)=qm*dx*dy
   where n,m = leftmost grid points and dx = x-n, dy = y-m
   ppart[m][n][0] = position x of particle n in tile m
   ppart[m][n][1] = position y of particle n in tile m
   q[k][j] = charge density at grid point j,k
   kpic = number of particles per tile
   qm = charge on particle, in units of e
   nppmx = maximum number of particles in tile
   idimp = size of phase space = 4
   mx/my = number of grids in sorting cell in x/y
   nxv = first dimension of charge array, must be >= nx+1
   nyv = second dimension of charge array, must be >= ny+1
   mx1 = (system length in x direction - 1)/mx + 1
   mxy1 = mx1*my1, where my1 = (system length in y direction - 1)/my + 1
local data                                                            */
#define MXV             33
#define MYV             33
   int noff, moff, npoff, npp, mxv;
   int i, j, k, nn, mm;
   float x, y, dxp, dyp, amx, amy;
   float sq[MXV*MYV];
/* float sq[(mx+1)*(my+1)]; */
   mxv = mx + 1;
/* error if local array is too small */
/* if ((mx >= MXV) || (my >= MYV))   */
/*    return;                        */
/* loop over tiles */
#pragma omp parallel for \
private(i,j,k,noff,moff,npp,npoff,nn,mm,x,y,dxp,dyp,amx,amy,sq)
   for (k = 0; k < mxy1; k++) {
      noff = k/mx1;
      moff = my*noff;
      noff = mx*(k - mx1*noff);
      npp = kpic[k];
      npoff = nppmx*k;
/* zero out local accumulator */
      for (j = 0; j < mxv*(my+1); j++) {
```

```
         sq[j] = 0.0f;
      }
/* loop over particles in tile */
      for (j = 0; j < npp; j++) {
/* find interpolation weights */
         x = ppart[idimp*(j+npoff)];
         y = ppart[1+idimp*(j+npoff)];
         nn = x;
         mm = y;
         dxp = qm*(x - (float) nn);
         dyp = y - (float) mm;
         nn = nn - noff + mxv*(mm - moff);
         amx = qm - dxp;
         amy = 1.0f - dyp;
/* deposit charge within tile to local accumulator */
         x = sq[nn] + amx*amy;
         y = sq[nn+1] + dxp*amy;
         sq[nn] = x;
         sq[nn+1] = y;
         nn += mxv;
         x = sq[nn] + amx*dyp;
         y = sq[nn+1] + dxp*dyp;
         sq[nn] = x;
         sq[nn+1] = y;
      }
/* deposit charge to interior points in global array */
      nn = nxv - noff;
      mm = nyv - moff;
      nn = mx < nn ? mx : nn;
      mm = my < mm ? my : mm;
      for (j = 1; j < mm; j++) {
         for (i = 1; i < nn; i++) {
            q[i+noff+nxv*(j+moff)] += sq[i+mxv*j];
         }
      }
/* deposit charge to edge points in global array */
      mm = nyv - moff;
      mm = my+1 < mm ? my+1 : mm;
      for (i = 1; i < nn; i++) {
#pragma omp atomic
         q[i+noff+nxv*moff] += sq[i];
         if (mm > my) {
#pragma omp atomic
            q[i+noff+nxv*(mm+moff-1)] += sq[i+mxv*(mm-1)];
         }
      }
      nn = nxv - noff;
      nn = mx+1 < nn ? mx+1 : nn;
      for (j = 0; j < mm; j++) {
#pragma omp atomic
         q[noff+nxv*(j+moff)] += sq[mxv*j];
         if (nn > mx) {
#pragma omp atomic
            q[nn+noff-1+nxv*(j+moff)] += sq[nn-1+mxv*j];
```

```
            }
        }
    }
    return;
#undef MXV
#undef MYV
}
```

```
/*--------------------------------------------------------------------*/
void cgrjppostf2l(float ppart[], float cu[], int kpic[], int ncl[],
                  int ihole[], float qm, float dt, float ci, int nppmx,
                  int idimp, int nx, int ny, int mx, int my, int nxv,
                  int nyv, int mx1, int mxy1, int ntmax, int *irc) {
/* for 2-1/2d code, this subroutine calculates particle current density
   using first-order linear interpolation for relativistic particles
   in addition, particle positions are advanced a half time-step
   with periodic boundary conditions.
   also determines list of particles which are leaving this tile
   OpenMP version using guard cells
   data deposited in tiles
   particles stored segmented array
   47 flops/particle, 1 divide, 1 sqrt, 17 loads, 14 stores
   input: all except ncl, ihole, irc,
   output: ppart, cu, ncl, ihole, irc
   current density is approximated by values at the nearest grid points
   cu(i,n,m)=qci*(1.-dx)*(1.-dy)
   cu(i,n+1,m)=qci*dx*(1.-dy)
   cu(i,n,m+1)=qci*(1.-dx)*dy
   cu(i,n+1,m+1)=qci*dx*dy
   where n,m = leftmost grid points and dx = x-n, dy = y-m
   and qci = qm*pi*gami, where i = x,y,z
   where gami = 1./sqrt(1.+sum(pi**2)*ci*ci)
   ppart[m][n][0] = position x of particle n in tile m
   ppart[m][n][1] = position y of particle n in tile m
   ppart[m][n][2] = x momentum of particle n in tile m
   ppart[m][n][3] = y momentum of particle n in tile m
   ppart[m][n][4] = z momentum of particle n in tile m
   cu[k][j][i] = ith component of current density at grid point j,k
   kpic[k] = number of particles in tile k
   ncl[k][i] = number of particles going to destination i, tile k
   ihole[k][:][0] = location of hole in array left by departing particle
   ihole[k][:][1] = destination of particle leaving hole
   ihole[k][0][0] = ih, number of holes left (error, if negative)
   qm = charge on particle, in units of e
   dt = time interval between successive calculations
   ci = reciprical of velocity of light
   nppmx = maximum number of particles in tile
   idimp = size of phase space = 5
   nx/ny = system length in x/y direction
   mx/my = number of grids in sorting cell in x/y
   nxv = first dimension of current array, must be >= nx+1
   nyv = second dimension of current array, must be >= ny+1
   mx1 = (system length in x direction - 1)/mx + 1
   mxy1 = mx1*my1, where my1 = (system length in y direction - 1)/my + 1
   ntmax = size of hole array for particles leaving tiles
   irc = maximum overflow, returned only if error occurs, when irc > 0
   optimized version
local data                                                            */
#define MXV             33
#define MYV             33
   int noff, moff, npoff, npp;
   int i, j, k, ih, nh, nn, mm, mxv3;
```

```
   float ci2, dxp, dyp, amx, amy;
   float x, y, dx, dy, vx, vy, vz, p2, gami;
   float anx, any, edgelx, edgely, edgerx, edgery;
   float scu[3*MXV*MYV];
/* float scu[3*(mx+1)*(my+1)]; */
   mxv3 = 3*(mx + 1);
   ci2 = ci*ci;
   anx = (float) nx;
   any = (float) ny;
/* error if local array is too small */
/* if ((mx >= MXV) || (my >= MYV))   */
/*    return;                        */
/* loop over tiles */
#pragma omp parallel for \
private(i,j,k,noff,moff,npp,npoff,nn,mm,ih,nh,x,y,dxp,dyp,amx,amy,dx, \
dy,vx,vy,vz,edgelx,edgely,edgerx,edgery,p2,gami,scu)
   for (k = 0; k < mxy1; k++) {
      noff = k/mx1;
      moff = my*noff;
      noff = mx*(k - mx1*noff);
      npp = kpic[k];
      npoff = nppmx*k;
      nn = nx - noff;
      nn = mx < nn ? mx : nn;
      mm = ny - moff;
      mm = my < mm ? my : mm;
      edgelx = noff;
      edgerx = noff + nn;
      edgely = moff;
      edgery = moff + mm;
      ih = 0;
      nh = 0;
      nn += 1;
      mm += 1;
/* zero out local accumulator */
      for (j = 0; j < mxv3*(my+1); j++) {
         scu[j] = 0.0f;
      }
/* clear counters */
      for (j = 0; j < 8; j++) {
         ncl[j+8*k] = 0;
      }
/* loop over particles in tile */
      for (j = 0; j < npp; j++) {
/* find interpolation weights */
         x = ppart[idimp*(j+npoff)];
         y = ppart[1+idimp*(j+npoff)];
         nn = x;
         mm = y;
         dxp = qm*(x - (float) nn);
         dyp = y - (float) mm;
/* find inverse gamma */
         vx = ppart[2+idimp*(j+npoff)];
         vy = ppart[3+idimp*(j+npoff)];
```

```
         vz = ppart[4+idimp*(j+npoff)];
         p2 = vx*vx + vy*vy + vz*vz;
         gami = 1.0/sqrtf(1.0 + p2*ci2);
/* calculate weights */
         nn = 3*(nn - noff) + mxv3*(mm - moff);
         amx = qm - dxp;
         amy = 1.0 - dyp;
/* deposit current */
         dx = amx*amy;
         dy = dxp*amy;
         vx *= gami;
         vy *= gami;
         vz *= gami;
         scu[nn] += vx*dx;
         scu[nn+1] += vy*dx;
         scu[nn+2] += vz*dx;
         dx = amx*dyp;
         mm = nn + 3;
         scu[mm] += vx*dy;
         scu[mm+1] += vy*dy;
         scu[mm+2] += vz*dy;
         dy = dxp*dyp;
         nn += mxv3;
         scu[nn] += vx*dx;
         scu[nn+1] += vy*dx;
         scu[nn+2] += vz*dx;
         mm = nn + 3;
         scu[mm] += vx*dy;
         scu[mm+1] += vy*dy;
         scu[mm+2] += vz*dy;
/* advance position half a time-step */
         dx = x + vx*dt;
         dy = y + vy*dt;
/* find particles going out of bounds */
         mm = 0;
/* count how many particles are going in each direction in ncl   */
/* save their address and destination in ihole                   */
/* use periodic boundary conditions and check for roundoff error */
/* mm = direction particle is going                              */
         if (dx >= edgerx) {
            if (dx >= anx)
               dx -= anx;
            mm = 2;
         }
         else if (dx < edgelx) {
            if (dx < 0.0f) {
               dx += anx;
               if (dx < anx)
                  mm = 1;
               else
                  dx = 0.0;
            }
            else {
               mm = 1;
```

```
            }
         }
         if (dy >= edgery) {
            if (dy >= any)
               dy -= any;
            mm += 6;
         }
         else if (dy < edgely) {
            if (dy < 0.0) {
               dy += any;
               if (dy < any)
                  mm += 3;
               else
                  dy = 0.0;
            }
            else {
               mm += 3;
            }
         }
/* set new position */
         ppart[idimp*(j+npoff)] = dx;
         ppart[1+idimp*(j+npoff)] = dy;
/* increment counters */
         if (mm > 0) {
            ncl[mm+8*k-1] += 1;
            ih += 1;
            if (ih <= ntmax) {
               ihole[2*(ih+(ntmax+1)*k)] = j + 1;
               ihole[1+2*(ih+(ntmax+1)*k)] = mm;
            }
            else {
               nh = 1;
            }
         }
      }
/* deposit current to interior points in global array */
      nn = nxv - noff;
      mm = nyv - moff;
      nn = mx < nn ? mx : nn;
      mm = my < mm ? my : mm;
      for (j = 0; j < mm; j++) {
         for (i = 0; i < nn; i++) {
            cu[3*(i+noff+nxv*(j+moff))] += scu[3*i+mxv3*j];
            cu[1+3*(i+noff+nxv*(j+moff))] += scu[1+3*i+mxv3*j];
            cu[2+3*(i+noff+nxv*(j+moff))] += scu[2+3*i+mxv3*j];
         }
      }
/* deposit current to edge points in global array */
      mm = nypmx - moffp;
      mm = my+1 < mm ? my+1 : mm;
      for (i = 1; i < nn; i++) {
#pragma omp atomic
         cu[3*(i+noffp+nxv*moffp)] += scu[3*i];
#pragma omp atomic
```

```
         cu[1+3*(i+noffp+nxv*moffp)] += scu[1+3*i];
#pragma omp atomic
         cu[2+3*(i+noffp+nxv*moffp)] += scu[2+3*i];
         if (mm > my) {
#pragma omp atomic
            cu[3*(i+noffp+nxv*(mm+moffp-1))] += scu[3*i+mxv3*(mm-1)];
#pragma omp atomic
            cu[1+3*(i+noffp+nxv*(mm+moffp-1))] += scu[1+3*i+mxv3*(mm-1)];
#pragma omp atomic
            cu[2+3*(i+noffp+nxv*(mm+moffp-1))] += scu[2+3*i+mxv3*(mm-1)];
         }
      }
      nn = nxv - noffp;
      nn = mx+1 < nn ? mx+1 : nn;
      for (j = 0; j < mm; j++) {
#pragma omp atomic
         cu[3*(noffp+nxv*(j+moffp))] += scu[mxv3*j];
#pragma omp atomic
         cu[1+3*(noffp+nxv*(j+moffp))] += scu[1+mxv3*j];
#pragma omp atomic
         cu[2+3*(noffp+nxv*(j+moffp))] += scu[2+mxv3*j];
         if (nn > mx) {
#pragma omp atomic
            cu[3*(nn+noffp-1+nxv*(j+moffp))] += scu[3*(nn-1)+mxv3*j];
#pragma omp atomic
            cu[1+3*(nn+noffp-1+nxv*(j+moffp))] += scu[1+3*(nn-1)+mxv3*j];
#pragma omp atomic
            cu[2+3*(nn+noffp-1+nxv*(j+moffp))] += scu[2+3*(nn-1)+mxv3*j];
         }
      }
/* set error and end of file flag */
/* ihole overflow */
      if (nh > 0) {
         *irc = ih;
         ih = -ih;
      }
      ihole[2*(ntmax+1)*k] = ih;
   }
   return;
#undef MXV
#undef MYV
}
```