```
/*--------------------------------------------------------------------*/
void cvgrbppushf23lt(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
   vectorizable/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][0][n] = position x of particle n in tile m
   ppart[m][1][n] = position y of particle n in tile m
   ppart[m][2][n] = x momentum of particle n in tile m
   ppart[m][3][n] = y momentum of particle n in tile m
   ppart[m][4][n] = 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 = reciprocal 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 = second dimension of field arrays, must be >= nx+1
   nyv = third 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
#define NPBLK           32
#define LVECT           4
#define N 4
   int noff, moff, npoff, npp, ipp, joff, nps;
   int i, j, k, m, ih, nh, nn, mm, nm, lxv;
   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, vx, vy, vz;
   float sfxy[N*MXV*MYV], sbxy[N*MXV*MYV];
/* float sfxy[N*(mx+1)*(my+1)], sbxy[N*(mx+1)*(my+1)]; */
/* scratch arrays */
   int n[NPBLK];
   float s1[NPBLK*LVECT], s2[NPBLK*LVECT], t[NPBLK*2];
   double sum1, sum2;
   lxv = mx + 1;
   qtmh = 0.5f*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,m,noff,moff,npp,npoff,ipp,joff,nps,nn,mm,nm,ih,nh,x,y,vx, \
vy,vz,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,n,s1,s2,t) \
reduction(+:sum2)
   for (k = 0; k < mxy1; k++) {
      noff = k/mx1;
      moff = my*noff;
      noff = mx*(k - mx1*noff);
      npp = kpic[k];
      npoff = idimp*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[N*(i+lxv*j)] = fxy[N*(i+noff+nxv*(j+moff))];
            sfxy[1+N*(i+lxv*j)] = fxy[1+N*(i+noff+nxv*(j+moff))];
            sfxy[2+N*(i+lxv*j)] = fxy[2+N*(i+noff+nxv*(j+moff))];
         }
      }
      for (j = 0; j < mm; j++) {
         for (i = 0; i < nn; i++) {
            sbxy[N*(i+lxv*j)] = bxy[N*(i+noff+nxv*(j+moff))];
            sbxy[1+N*(i+lxv*j)] = bxy[1+N*(i+noff+nxv*(j+moff))];
            sbxy[2+N*(i+lxv*j)] = bxy[2+N*(i+noff+nxv*(j+moff))];
         }
      }
/* clear counters */
      for (j = 0; j < 8; j++) {
         ncl[j+8*k] = 0;
      }
      sum1 = 0.0;
      ipp = npp/NPBLK;
/* outer loop over number of full blocks */
      for (m = 0; m < ipp; m++) {
         joff = NPBLK*m;
/* inner loop over particles in block */
         for (j = 0; j < NPBLK; j++) {
/* find interpolation weights */
            x = ppart[j+joff+npoff];
            y = ppart[j+joff+nppmx+npoff];
            nn = x;
            mm = y;
            dxp = x - (float) nn;
            dyp = y - (float) mm;
            n[j] = N*(nn - noff + lxv*(mm - moff));
            amx = 1.0f - dxp;
            amy = 1.0f - dyp;
            s1[j] = amx*amy;
            s1[j+NPBLK] = dxp*amy;
            s1[j+2*NPBLK] = amx*dyp;
            s1[j+3*NPBLK] = dxp*dyp;
            t[j] = x;
            t[j+NPBLK] = y;
         }
/* find acceleration */
         for (j = 0; j < NPBLK; j++) {
            nn = n[j];
            mm = nn + N*(lxv - 2);
            dx = 0.0f;
            dy = 0.0f;
            dz = 0.0f;
            ox = 0.0f;
            oy = 0.0f;
            oz = 0.0f;
#pragma ivdep
            for (i = 0; i < LVECT; i++) {
               if (i > 1)
                  nn = mm;
               dx += sfxy[N*i+nn]*s1[j+NPBLK*i];
               dy += sfxy[1+N*i+nn]*s1[j+NPBLK*i];
               dz += sfxy[2+N*i+nn]*s1[j+NPBLK*i];
               ox += sbxy[N*i+nn]*s1[j+NPBLK*i];
               oy += sbxy[1+N*i+nn]*s1[j+NPBLK*i];
               oz += sbxy[2+N*i+nn]*s1[j+NPBLK*i];
            }
            s1[j] = dx;
            s1[j+NPBLK] = dy;
            s1[j+2*NPBLK] = dz;
            s2[j] = ox;
            s2[j+NPBLK] = oy;
            s2[j+2*NPBLK] = oz;
         }
/* new momentum */
         for (j = 0; j < NPBLK; j++) {
            x = t[j];
            y = t[j+NPBLK];
/* calculate half impulse */
            dx = qtmh*s1[j];
            dy = qtmh*s1[j+NPBLK];
            dz = qtmh*s1[j+2*NPBLK];
/* half acceleration */
            acx = ppart[j+joff+2*nppmx+npoff] + dx;
            acy = ppart[j+joff+3*nppmx+npoff] + dy;
            acz = ppart[j+joff+4*nppmx+npoff] + dz;
/* find inverse gamma */
            p2 = acx*acx + acy*acy + acz*acz;
            gami = 1.0f/sqrtf(1.0f + p2*ci2);
/* renormalize magnetic field */
            qtmg = qtmh*gami;
/* time-centered kinetic energy */
            sum1 += gami*p2/(1.0f + gami);
/* calculate cyclotron frequency */
            omxt = qtmg*s2[j];
            omyt = qtmg*s2[j+NPBLK];
            omzt = qtmg*s2[j+2*NPBLK];
/* calculate rotation matrix */
            omt = omxt*omxt + omyt*omyt + omzt*omzt;
            anorm = 2.0f/(1.0f + omt);
            omt = 0.5f*(1.0f - 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 */
            vx = (rot1*acx + rot2*acy + rot3*acz)*anorm + dx;
            vy = (rot4*acx + rot5*acy + rot6*acz)*anorm + dy;
            vz = (rot7*acx + rot8*acy + rot9*acz)*anorm + dz;
/* update inverse gamma */
            p2 = vx*vx + vy*vy + vz*vz;
            dtg = dtc/sqrtf(1.0f + p2*ci2);
/* new position */
            s1[j] = x + vx*dtg;
            s1[j+NPBLK] = y + vy*dtg;
            s2[j] = vx;
            s2[j+NPBLK] = vy;
            s2[j+2*NPBLK] = vz;
         }
/* check boundary conditions */
#pragma novector
         for (j = 0; j < NPBLK; j++) {
            dx = s1[j];
            dy = s1[j+NPBLK];
/* 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[j+joff+npoff] = dx;
            ppart[j+joff+nppmx+npoff] = dy;
/* set new momentum */
            ppart[j+joff+2*nppmx+npoff] = s2[j];
            ppart[j+joff+3*nppmx+npoff] = s2[j+NPBLK];
            ppart[j+joff+4*nppmx+npoff] = s2[j+2*NPBLK];
```

```
/* increment counters */
            if (mm > 0) {
               ncl[mm+8*k-1] += 1;
               ih += 1;
               if (ih <= ntmax) {
                  ihole[2*(ih+(ntmax+1)*k)] = j + joff + 1;
                  ihole[1+2*(ih+(ntmax+1)*k)] = mm;
               }
               else {
                  nh = 1;
               }
            }
         }
      }
      nps = NPBLK*ipp;
/* loop over remaining particles */
      for (j = nps; j < npp; j++) {
/* find interpolation weights */
         x = ppart[j+npoff];
         y = ppart[j+nppmx+npoff];
         nn = x;
         mm = y;
         dxp = x - (float) nn;
         dyp = y - (float) mm;
         nm = N*(nn - noff + lxv*(mm - moff));
         amx = 1.0f - dxp;
         amy = 1.0f - dyp;
/* find electric field */
         nn = nm;
         dx = amx*sfxy[nn];
         dy = amx*sfxy[nn+1];
         dz = amx*sfxy[nn+2];
         mm = nn + N;
         dx = amy*(dxp*sfxy[mm] + dx);
         dy = amy*(dxp*sfxy[mm+1] + dy);
         dz = amy*(dxp*sfxy[mm+2] + dz);
         nn += N*lxv;
         acx = amx*sfxy[nn];
         acy = amx*sfxy[nn+1];
         acz = amx*sfxy[nn+2];
         mm = nn + N;
         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 + N;
         ox = amy*(dxp*sbxy[mm] + ox);
         oy = amy*(dxp*sbxy[mm+1] + oy);
         oz = amy*(dxp*sbxy[mm+2] + oz);
         nn += N*lxv;
         acx = amx*sbxy[nn];
         acy = amx*sbxy[nn+1];
         acz = amx*sbxy[nn+2];
         mm = nn + N;
         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[j+2*nppmx+npoff] + dx;
         acy = ppart[j+3*nppmx+npoff] + dy;
         acz = ppart[j+4*nppmx+npoff] + dz;
/* find inverse gamma */
         p2 = acx*acx + acy*acy + acz*acz;
         gami = 1.0f/sqrtf(1.0f + p2*ci2);
/* renormalize magnetic field */
         qtmg = qtmh*gami;
/* time-centered kinetic energy */
         sum1 += gami*p2/(1.0f + gami);
/* calculate cyclotron frequency */
         omxt = qtmg*ox;
         omyt = qtmg*oy;
         omzt = qtmg*oz;
/* calculate rotation matrix */
         omt = omxt*omxt + omyt*omyt + omzt*omzt;
         anorm = 2.0f/(1.0f + omt);
         omt = 0.5f*(1.0f - 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 */
         vx = (rot1*acx + rot2*acy + rot3*acz)*anorm + dx;
         vy = (rot4*acx + rot5*acy + rot6*acz)*anorm + dy;
         vz = (rot7*acx + rot8*acy + rot9*acz)*anorm + dz;
/* update inverse gamma */
         p2 = vx*vx + vy*vy + vz*vz;
         dtg = dtc/sqrtf(1.0f + p2*ci2);
/* new position */
         dx = x + vx*dtg;
         dy = y + vy*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[j+npoff] = dx;
         ppart[j+nppmx+npoff] = dy;
/* set new momentum */
         ppart[j+2*nppmx+npoff] = vx;
         ppart[j+3*nppmx+npoff] = vy;
         ppart[j+4*nppmx+npoff] = vz;
/* 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 N
#undef LVECT
#undef NPBLK
#undef MXV
#undef MYV
}
```

```
/*--------------------------------------------------------------------*/
void cvgppost2lt(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
   vectorizable/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][0][n] = position x of particle n in tile m
   ppart[m][1][n] = 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
#define NPBLK             32
#define LVECT             4
   int noff, moff, npoff, npp, ipp, joff, nps;
   int i, j, k, m, nn, mm, lxv;
   float x, y, dxp, dyp, amx, amy;
   float sq[MXV*MYV];
/* float sq[(mx+1)*(my+1)]; */
/* scratch arrays */
   int n[NPBLK];
   float s[NPBLK*LVECT];
   lxv = 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,m,noff,moff,npp,npoff,ipp,joff,nps,nn,mm,x,y,dxp,dyp, \
amx,amy,sq,n,s)
   for (k = 0; k < mxy1; k++) {
      noff = k/mx1;
      moff = my*noff;
      noff = mx*(k - mx1*noff);
      npp = kpic[k];
      npoff = idimp*nppmx*k;
/* zero out local accumulator */
      for (j = 0; j < lxv*(my+1); j++) {
         sq[j] = 0.0f;
      }
/* loop over particles in tile */
      ipp = npp/NPBLK;
/* outer loop over number of full blocks */
      for (m = 0; m < ipp; m++) {
         joff = NPBLK*m;
/* inner loop over particles in block */
         for (j = 0; j < NPBLK; j++) {
/* find interpolation weights */
            x = ppart[j+joff+npoff];
            y = ppart[j+joff+nppmx+npoff];
            nn = x;
            mm = y;
            dxp = qm*(x - (float) nn);
            dyp = y - (float) mm;
            n[j] = nn - noff + lxv*(mm - moff);
            amx = qm - dxp;
            amy = 1.0f - dyp;
            s[j] = amx*amy;
            s[j+NPBLK] = dxp*amy;
            s[j+2*NPBLK] = amx*dyp;
            s[j+3*NPBLK] = dxp*dyp;
        }
/* deposit charge within tile to local accumulator */
         for (j = 0; j < NPBLK; j++) {
            nn = n[j];
            mm = nn + lxv - 2;
#pragma ivdep
            for (i = 0; i < LVECT; i++) {
               if (i > 1)
                  nn = mm;
               sq[i+nn] += s[j+NPBLK*i];
            }
         }
      }
      nps = NPBLK*ipp;
/* loop over remaining particles */
      for (j = nps; j < npp; j++) {
/* find interpolation weights */
         x = ppart[j+npoff];
         y = ppart[j+nppmx+npoff];
         nn = x;
         mm = y;
         dxp = qm*(x - (float) nn);
         dyp = y - (float) mm;
         nn = nn - noff + lxv*(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 += lxv;
         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+lxv*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+lxv*(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[lxv*j];
         if (nn > mx) {
#pragma omp atomic
            q[nn+noff-1+nxv*(j+moff)] += sq[nn-1+lxv*j];
         }
      }
   }
   return;
#undef LVECT
#undef NPBLK
#undef MXV
#undef MYV
}
```

```
/*--------------------------------------------------------------------*/
void cvgrjppostf2lt(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
   vectorizable/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][0][n] = position x of particle n in tile m
   ppart[m][1][n] = position y of particle n in tile m
   ppart[m][2][n] = x momentum of particle n in tile m
   ppart[m][3][n] = y momentum of particle n in tile m
   ppart[m][4][n] = 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 = reciprocal 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 = second dimension of current array, must be >= nx+1
   nyv = third 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
```

```
#define NPBLK             32
#define LVECT             4
#define N 4
   int noff, moff, npoff, npp, lxv;
   int i, j, k, m, ih, nh, ipp, joff, nps, nn, mm;
   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[N*MXV*MYV];
/* float scu[N*(mx+1)*(my+1)]; */
/* scratch arrays */
   int n[NPBLK];
   float s1[NPBLK*LVECT], s2[NPBLK*LVECT], t[NPBLK*2];
   lxv = 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,m,noff,moff,npp,npoff,ipp,joff,nps,nn,mm,ih,nh,x,y,dxp, \
dyp,amx,amy,dx,dy,vx,vy,vz,edgelx,edgely,edgerx,edgery,p2,gami,scu,n, \
s1,s2,t)
   for (k = 0; k < mxy1; k++) {
      noff = k/mx1;
      moff = my*noff;
      noff = mx*(k - mx1*noff);
      npp = kpic[k];
      npoff = idimp*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 < N*lxv*(my+1); j++) {
         scu[j] = 0.0f;
      }
/* clear counters */
      for (j = 0; j < 8; j++) {
         ncl[j+8*k] = 0;
      }
      ipp = npp/NPBLK;
/* outer loop over number of full blocks */
      for (m = 0; m < ipp; m++) {
         joff = NPBLK*m;
/* inner loop over particles in block */
         for (j = 0; j < NPBLK; j++) {
/* find interpolation weights */
            x = ppart[j+joff+npoff];
            y = ppart[j+joff+nppmx+npoff];
            nn = x;
            mm = y;
            dxp = qm*(x - (float) nn);
            dyp = y - (float) mm;
            n[j] = N*(nn - noff + lxv*(mm - moff));
            amx = qm - dxp;
            amy = 1.0f - dyp;
            s1[j] = amx*amy;
            s1[j+NPBLK] = dxp*amy;
            s1[j+2*NPBLK] = amx*dyp;
            s1[j+3*NPBLK] = dxp*dyp;
            t[j] = x;
            t[j+NPBLK] = y;
/* find inverse gamma */
            vx = ppart[j+joff+2*nppmx+npoff];
            vy = ppart[j+joff+3*nppmx+npoff];
            vz = ppart[j+joff+4*nppmx+npoff];
            p2 = vx*vx + vy*vy + vz*vz;
            gami = 1.0f/sqrtf(1.0f + p2*ci2);
            s2[j] = vx*gami;
            s2[j+NPBLK] = vy*gami;
            s2[j+2*NPBLK] = vz*gami;
         }
/* deposit current */
         for (j = 0; j < NPBLK; j++) {
            nn = n[j];
            mm = nn + N*(lxv - 2);
            vx = s2[j];
            vy = s2[j+NPBLK];
            vz = s2[j+2*NPBLK];
#pragma ivdep
            for (i = 0; i < LVECT; i++) {
               if (i > 1)
                  nn = mm;
               scu[N*i+nn] += vx*s1[j+NPBLK*i];
               scu[1+N*i+nn] += vy*s1[j+NPBLK*i];
               scu[2+N*i+nn] += vz*s1[j+NPBLK*i];
            }
         }
/* advance position half a time-step */
         for (j = 0; j < NPBLK; j++) {
            dx = t[j] + s2[j]*dt;
            dy = t[j+NPBLK] + s2[j+NPBLK]*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[j+joff+npoff] = dx;
            ppart[j+joff+nppmx+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 + joff + 1;
                  ihole[1+2*(ih+(ntmax+1)*k)] = mm;
               }
               else {
                  nh = 1;
               }
            }
         }
      }
      nps = NPBLK*ipp;
/* loop over remaining particles */
```

```
      for (j = nps; j < npp; j++) {
/* find interpolation weights */
         x = ppart[j+npoff];
         y = ppart[j+nppmx+npoff];
         nn = x;
         mm = y;
         dxp = qm*(x - (float) nn);
         dyp = y - (float) mm;
/* find inverse gamma */
         vx = ppart[j+2*nppmx+npoff];
         vy = ppart[j+3*nppmx+npoff];
         vz = ppart[j+4*nppmx+npoff];
         p2 = vx*vx + vy*vy + vz*vz;
         gami = 1.0f/sqrtf(1.0f + p2*ci2);
/* calculate weights */
         nn = N*(nn - noff + lxv*(mm - moff));
         amx = qm - dxp;
         amy = 1.0f - 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 + N;
         scu[mm] += vx*dy;
         scu[mm+1] += vy*dy;
         scu[mm+2] += vz*dy;
         dy = dxp*dyp;
         nn += N*lxv;
         scu[nn] += vx*dx;
         scu[nn+1] += vy*dx;
         scu[nn+2] += vz*dx;
         mm = nn + N;
         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[j+npoff] = dx;
         ppart[j+nppmx+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 = 1; j < mm; j++) {
         for (i = 1; i < nn; i++) {
            cu[N*(i+noff+nxv*(j+moff))] += scu[N*(i+lxv*j)];
            cu[1+N*(i+noff+nxv*(j+moff))] += scu[1+N*(i+lxv*j)];
            cu[2+N*(i+noff+nxv*(j+moff))] += scu[2+N*(i+lxv*j)];
         }
      }
/* deposit current 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
         cu[N*(i+noff+nxv*moff)] += scu[N*i];
#pragma omp atomic
         cu[1+N*(i+noff+nxv*moff)] += scu[1+N*i];
#pragma omp atomic
         cu[2+N*(i+noff+nxv*moff)] += scu[2+N*i];
         if (mm > my) {
#pragma omp atomic
            cu[N*(i+noff+nxv*(mm+moff-1))] += scu[N*(i+lxv*(mm-1))];
#pragma omp atomic
            cu[1+N*(i+noff+nxv*(mm+moff-1))] += scu[1+N*(i+lxv*(mm-1))];
#pragma omp atomic
            cu[2+N*(i+noff+nxv*(mm+moff-1))] += scu[2+N*(i+lxv*(mm-1))];
         }
      }
      nn = nxv - noff;
      nn = mx+1 < nn ? mx+1 : nn;
      for (j = 0; j < mm; j++) {
#pragma omp atomic
         cu[N*(noff+nxv*(j+moff))] += scu[N*lxv*j];
#pragma omp atomic
         cu[1+N*(noff+nxv*(j+moff))] += scu[1+N*lxv*j];
#pragma omp atomic
         cu[2+N*(noff+nxv*(j+moff))] += scu[2+N*lxv*j];
         if (nn > mx) {
#pragma omp atomic
            cu[N*(nn+noff-1+nxv*(j+moff))] += scu[N*((nn-1)+lxv*j)];
#pragma omp atomic
            cu[1+N*(nn+noff-1+nxv*(j+moff))] += scu[1+N*((nn-1)+lxv*j)];
#pragma omp atomic
            cu[2+N*(nn+noff-1+nxv*(j+moff))] += scu[2+N*((nn-1)+lxv*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 N
#undef LVECT
#undef NPBLK
#undef MXV
```

```
#undef MYV
}
```