```
/*--------------------------------------------------------------------*/
void cppgrbpush23l(float part[], float fxy[], float bxy[],
                   float edges[], int npp, int noff, int ihole[],
                   float qbm, float dt, float dtc, float ci, float *ek,
                   int nx, int ny, int idimp, int npmax, int nxv,
                   int nypmx, int idps, int ntmax, int ipbc) {
/* 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
   Using the Boris Mover.
   scalar version using guard cells, for distributed data
   also determines list of particles which are leaving this processor
   129 flops/particle, 4 divides, 2 sqrts, 25 loads, 5 stores
   input: all except ihole, output: part, ihole, ek
   momentum equations used are:
   px(t+dt/2) = rot[0]*(px(t-dt/2) + .5*(q/m)*fx(x(t),y(t))*dt) +
      rot[1]*(py(t-dt/2) + .5*(q/m)*fy(x(t),y(t))*dt) +
      rot[2]*(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[3]*(px(t-dt/2) + .5*(q/m)*fx(x(t),y(t))*dt) +
      rot[4]*(py(t-dt/2) + .5*(q/m)*fy(x(t),y(t))*dt) +
      rot[5]*(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[6]*(px(t-dt/2) + .5*(q/m)*fx(x(t),y(t))*dt) +
      rot[7]*(py(t-dt/2) + .5*(q/m)*fy(x(t),y(t))*dt) +
      rot[8]*(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)), 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)
```

```
   part[n][0] = position x of particle n in partition
   part[n][1] = position y of particle n in partition
   part[n][2] = momentum px of particle n in partition
   part[n][3] = momentum py of particle n in partition
   part[n][4] = momentum pz of particle n in partition
   fxy[k][j][0] = x component of force/charge at grid (j,kk)
   fxy[k][j][1] = y component of force/charge at grid (j,kk)
   fxy[k][j][2] = z component of force/charge at grid (j,kk)
   that is, convolution of electric field over particle shape
   bxy[k][j][0] = x component of magnetic field at grid (j,kk)
   bxy[k][j][1] = y component of magnetic field at grid (j,kk)
   bxy[k][j][2] = z component of magnetic field at grid (j,kk)
   that is, the convolution of magnetic field over particle shape
   where kk = k + noff
   edges[0:1] = lower:upper boundary of particle partition
   npp = number of particles in partition
   noff = lowermost global gridpoint in particle partition.
   ihole = location of hole left in particle arrays
   ihole(1) = 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)
   nx/ny = system length in x/y direction
   idimp = size of phase space = 5
   npmax = maximum number of particles in each partition
   nxv = first dimension of field arrays, must be >= nx+1
   nypmx = maximum size of particle partition, including guard cells.
   idps = number of partition boundaries
   ntmax = size of hole array for particles leaving processors
   ipbc = particle boundary condition = (0,1,2,3) =
   (none,2d periodic,2d reflecting,mixed reflecting/periodic)
local data                                                          */
   int mnoff, j, nn, mm, np, mp, ih, nh, nxv3;
   float qtmh, ci2, edgelx, edgely, edgerx, edgery, dxp, dyp, amx, amy;
   float dx, dy, dz, ox, oy, oz, acx, acy, acz, p2, gami, qtmg, dtg;
   float omxt, omyt, omzt, omt, anorm;
   float rot1, rot2, rot3, rot4, rot5, rot6, rot7, rot8, rot9;
   double sum1;
   nxv3 = 3*nxv;
   qtmh = 0.5*qbm*dt;
   ci2 = ci*ci;
   sum1 = 0.0;
/* set boundary values */
   edgelx = 0.0;
   edgely = 1.0;
   edgerx = (float) nx;
   edgery = (float) (ny-1);
   if ((ipbc==2) || (ipbc==3)) {
      edgelx = 1.0;
```

```
      edgerx = (float) (nx-1);
   }
   mnoff = noff;
   ih = 0;
   nh = 0;
   for (j = 0; j < npp; j++) {
/* find interpolation weights */
      nn = part[idimp*j];
      mm = part[1+idimp*j];
      dxp = part[idimp*j] - (float) nn;
      dyp = part[1+idimp*j] - (float) mm;
      nn = 3*nn;
      mm = nxv3*(mm - mnoff);
      amx = 1.0 - dxp;
      mp = mm + nxv3;
      amy = 1.0 - dyp;
      np = nn + 3;
/* find electric field */
      dx = dyp*(dxp*fxy[np+mp] + amx*fxy[nn+mp])
         + amy*(dxp*fxy[np+mm] + amx*fxy[nn+mm]);
      dy = dyp*(dxp*fxy[1+np+mp] + amx*fxy[1+nn+mp])
         + amy*(dxp*fxy[1+np+mm] + amx*fxy[1+nn+mm]);
      dz = dyp*(dxp*fxy[2+np+mp] + amx*fxy[2+nn+mp])
         + amy*(dxp*fxy[2+np+mm] + amx*fxy[2+nn+mm]);
/* calculate half impulse */
      dx *= qtmh;
      dy *= qtmh;
      dz *= qtmh;
/* half acceleration */
      acx = part[2+idimp*j] + dx;
      acy = part[3+idimp*j] + dy;
      acz = part[4+idimp*j] + dz;
/* find inverse gamma */
      p2 = acx*acx + acy*acy + acz*acz;
      gami = 1.0/sqrtf(1.0 + p2*ci2);
/* find magnetic field */
      ox = dyp*(dxp*bxy[np+mp] + amx*bxy[nn+mp])
         + amy*(dxp*bxy[np+mm] + amx*bxy[nn+mm]);
      oy = dyp*(dxp*bxy[1+np+mp] + amx*bxy[1+nn+mp])
         + amy*(dxp*bxy[1+np+mm] + amx*bxy[1+nn+mm]);
      oz = dyp*(dxp*bxy[2+np+mp] + amx*bxy[2+nn+mp])
         + amy*(dxp*bxy[2+np+mm] + amx*bxy[2+nn+mm]);
/* 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;
      part[2+idimp*j] = dx;
      part[3+idimp*j] = dy;
      part[4+idimp*j] = dz;
/* update inverse gamma */
      p2 = dx*dx + dy*dy + dz*dz;
      dtg = dtc/sqrtf(1.0 + p2*ci2);
/* new position */
      dx = part[idimp*j] + dx*dtg;
      dy = part[1+idimp*j] + dy*dtg;
/* periodic boundary conditions in x */
      if (ipbc==1) {
         if (dx < edgelx) dx += edgerx;
         if (dx >= edgerx) dx -= edgerx;
      }
/* reflecting boundary conditions */
      else if (ipbc==2) {
         if ((dx < edgelx) || (dx >= edgerx)) {
            dx = part[idimp*j];
            part[2+idimp*j] = -part[2+idimp*j];
         }
         if ((dy < edgely) || (dy >= edgery)) {
            dy = part[1+idimp*j];
            part[3+idimp*j] = -part[3+idimp*j];
         }
      }
/* mixed reflecting/periodic boundary conditions */
      else if (ipbc==3) {
         if ((dx < edgelx) || (dx >= edgerx)) {
            dx = part[idimp*j];
            part[2+idimp*j] = -part[2+idimp*j];
         }
      }
/* find particles out of bounds */
      if ((dy < edges[0]) || (dy >= edges[1])) {
         if (ih < ntmax)
            ihole[ih+1] = j + 1;
         else
            nh = 1;
```

```
         ih += 1;
      }
/* set new position */
      part[idimp*j] = dx;
      part[1+idimp*j] = dy;
   }
/* set end of file flag */
   if (nh > 0)
      ih = -ih;
   ihole[0] = ih;
/* normalize kinetic energy */
   *ek += sum1;
   return;
}
```

```
/*--------------------------------------------------------------------*/
void cppgpost2l(float part[], float q[], int npp, int noff, float qm,
                int idimp, int npmax, int nxv, int nypmx) {
/* for 2d code, this subroutine calculates particle charge density
   using first-order linear interpolation, periodic boundaries
   scalar version using guard cells, for distributed data
   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
   part[n][0] = position x of particle n in partition
   part[n][1] = position y of particle n in partition
   q[k][j] = charge density at grid point (j,kk),
   where kk = k + noff
   npp = number of particles in partition
   noff = lowermost global gridpoint in particle partition.
   qm = charge on particle, in units of e
   idimp = size of phase space = 4
   npmax = maximum number of particles in each partition
   nxv = first dimension of charge array, must be >= nx+1
   nypmx = maximum size of particle partition, including guard cells.
local data                                                            */
   int  mnoff, j, nn, np, mm, mp;
   float dxp, dyp, amx, amy;
   mnoff = noff;
   for (j = 0; j < npp; j++) {
/* find interpolation weights */
      nn = part[idimp*j];
      mm = part[1+idimp*j];
      dxp = qm*(part[idimp*j] - (float) nn);
      dyp = part[1+idimp*j] - (float) mm;
      mm = nxv*(mm - mnoff);
      amx = qm - dxp;
      mp = mm + nxv;
      amy = 1.0 - dyp;
      np = nn + 1;
/* deposit charge */
      q[np+mp] += dxp*dyp;
      q[nn+mp] += amx*dyp;
      q[np+mm] += dxp*amy;
      q[nn+mm] += amx*amy;
   }
   return;
}
```

```
/*--------------------------------------------------------------------*/
void cppgrjpost2l(float part[], float cu[], float edges[], int npp,
                  int noff, int ihole[], float qm, float dt, float ci,
                  int nx, int ny, int idimp, int npmax, int nxv,
                  int nypmx, int idps, int ntmax, int ipbc) {
/* 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
   also determines list of particles which are leaving this processor
   scalar version using guard cells, for distributed data
   45 flops/particle, 1 divide, 1 sqrt, 17 loads, 14 stores
   input: all except ihole, output: part, ihole, cu
   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)
   part[n][0] = position x of particle n in partition
   part[n][1] = position y of particle n in partition
   part[n][2] = momentum px of particle n in partition
   part[n][3] = momentum py of particle n in partition
   part[n][4] = momentum pz of particle n in partition
   cu[k][j][i] = ith component of current density at grid point j,kk,
   where kk = k + noff
   edges[0:1] = lower:upper boundary of particle partition
   npp = number of particles in partition
   noff = lowermost global gridpoint in particle partition.
   ihole = location of hole left in particle arrays
   ihole[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
   nx/ny = system length in x/y direction
   idimp = size of phase space = 5
   npmax = maximum number of particles in each partition
   nxv = first dimension of current array, must be >= nx+1
   nypmx = maximum size of particle partition, including guard cells.
   idps = number of partition boundaries
   ntmax =  size of hole array for particles leaving processors
   ipbc = particle boundary condition = (0,1,2,3) =
   (none,2d periodic,2d reflecting,mixed reflecting/periodic)
local data                                                          */
   int mnoff, j, nn, mm, np, mp, ih, nh, nxv3;
   float ci2, edgelx, edgely, edgerx, edgery, dxp, dyp, amx, amy;
   float dx, dy, vx, vy, vz, p2, gami;
   nxv3 = 3*nxv;
   ci2 = ci*ci;
/* set boundary values */
   edgelx = 0.0;
   edgely = 1.0;
   edgerx = (float) nx;
```

```
   edgery = (float) (ny-1);
   if ((ipbc==2) || (ipbc==3)) {
      edgelx = 1.0;
      edgerx = (float) (nx-1);
   }
   mnoff = noff;
   ih = 0;
   nh = 0;
   for (j = 0; j < npp; j++) {
/* find interpolation weights */
      nn = part[idimp*j];
      mm = part[1+idimp*j];
      dxp = qm*(part[idimp*j] - (float) nn);
      dyp = part[1+idimp*j] - (float) mm;
/* find inverse gamma */
      vx = part[2+idimp*j];
      vy = part[3+idimp*j];
      vz = part[4+idimp*j];
      p2 = vx*vx + vy*vy + vz*vz;
      gami = 1.0/sqrtf(1.0 + p2*ci2);
/* calculate weights */
      nn = 3*nn;
      mm = nxv3*(mm - mnoff);
      amx = qm - dxp;
      mp = mm + nxv3;
      amy = 1.0 - dyp;
      np = nn + 3;
/* deposit current */
      dx = dxp*dyp;
      dy = amx*dyp;
      vx *= gami;
      vy *= gami;
      vz *= gami;
      cu[np+mp] += vx*dx;
      cu[1+np+mp] += vy*dx;
      cu[2+np+mp] += vz*dx;
      dx = dxp*amy;
      cu[nn+mp] += vx*dy;
      cu[1+nn+mp] += vy*dy;
      cu[2+nn+mp] += vz*dy;
      dy = amx*amy;
      cu[np+mm] += vx*dx;
      cu[1+np+mm] += vy*dx;
      cu[2+np+mm] += vz*dx;
      cu[nn+mm] += vx*dy;
      cu[1+nn+mm] += vy*dy;
      cu[2+nn+mm] += vz*dy;
/* advance position half a time-step */
      dx = part[idimp*j] + vx*dt;
      dy = part[1+idimp*j] + vy*dt;
/* periodic boundary conditions in x */
      if (ipbc==1) {
         if (dx < edgelx) dx += edgerx;
         if (dx >= edgerx) dx -= edgerx;
```

```
      }
/* reflecting boundary conditions */
      else if (ipbc==2) {
         if ((dx < edgelx) || (dx >= edgerx)) {
            dx = part[idimp*j];
            part[2+idimp*j] = -part[2+idimp*j];
         }
         if ((dy < edgely) || (dy >= edgery)) {
            dy = part[1+idimp*j];
            part[3+idimp*j] = -part[3+idimp*j];
         }
      }
/* mixed reflecting/periodic boundary conditions */
      else if (ipbc==3) {
         if ((dx < edgelx) || (dx >= edgerx)) {
            dx = part[idimp*j];
            part[2+idimp*j] = -part[2+idimp*j];
         }
      }
/* find particles out of bounds */
      if ((dy < edges[0]) || (dy >= edges[1])) {
         if (ih < ntmax)
            ihole[ih+1] = j + 1;
         else
            nh = 1;
         ih += 1;
      }
/* set new position */
      part[idimp*j] = dx;
      part[1+idimp*j] = dy;
   }
/* set end of file flag */
   if (nh > 0)
      ih = -ih;
   ihole[0] = ih;
   return;
}
```