Minh Tran

Assignment 5 CSCI 596: Parallel Molecular Dynamics

I. Source code hmd.c

The new program hmd.c is modifed from pmd_irecv.c by inserting proper OpenMP-related messaging lines of code while taking advantage of exisitng modifications for MPI. The changes are marked by //###

```
/*----------------------------------------------------------------------
Program pmd.c performs parallel molecular-dynamics for Lennard-Jones
systems using the Message Passing Interface (MPI) standard.
----------------------------------------------------------------------*/
#include "hmd.h" // ###

/*--------------------------------------------------------------------*/
int main(int argc, char **argv) {
/*--------------------------------------------------------------------*/
 double cpu1;

 MPI_Init(&argc,&argv); /* Initialize the MPI environment */
 MPI_Comm_rank(MPI_COMM_WORLD, &sid);  /* My processor ID */
 /* Vector index of this processor */
 vid[0] = sid/(vproc[1]*vproc[2]);
 vid[1] = (sid/vproc[2])%vproc[1];
 vid[2] = sid%vproc[2];

 omp_set_num_threads(nthrd); // ###

 init_params();
 set_topology();
 init_conf();
 atom_copy();
 compute_accel(); /* Computes initial accelerations */

 cpu1 = MPI_Wtime();
 for (stepCount=1; stepCount<=StepLimit; stepCount++) {
  single_step();
  if (stepCount%StepAvg == 0) eval_props();
 }
 cpu = MPI_Wtime() - cpu1;
 if (sid == 0) printf("CPU & COMT = %le %le\n",cpu,comt);

 MPI_Finalize(); /* Clean up the MPI environment */
 return 0;
}

/*----------------------------------------------------------------------*/
void init_params() {
/*------------------------------------------------------------------------
Initializes parameters.
------------------------------------------------------------------------*/
 int a;
 double rr,ri2,ri6,r1;
 FILE *fp;

 /* Read control parameters */
 fp = fopen("hmd.in","r"); //###
 fscanf(fp,"%d%d%d",&InitUcell[0],&InitUcell[1],&InitUcell[2]);
 fscanf(fp,"%le",&Density);
 fscanf(fp,"%le",&InitTemp);
 fscanf(fp,"%le",&DeltaT);
 fscanf(fp,"%d",&StepLimit);
 fscanf(fp,"%d",&StepAvg);
 fclose(fp);

 /* Compute basic parameters */
 DeltaTH = 0.5*DeltaT;
 for (a=0; a<3; a++) al[a] = InitUcell[a]/pow(Density/4.0,1.0/3.0);
 if (sid == 0) printf("al = %e %e %e\n",al[0],al[1],al[2]);

 /* Compute the # of cells for linked cell lists */
 // lc is the number of linked-list cells in each direction
 // RCUT is the interaction cut of length
 // al is bulk length
 for (a=0; a<3; a++) {
   // to guarantee interaction of cells reside at most in the neighboring cells
   lc[a] = al[a]/RCUT; // cell size has to be at least greater than potential cutoff

   // size of cell block that each threadh is assigned ###
   thbk[a] = lc[a]/vthrd[a];

   // # of cells =  integrer multiple of the # of threads ###
   // lc[a] will be under-estimated (due to module operation above)
```

```
    lc[a] = thbk[a]*vthrd[a];

    // linked-list cell length: rc[a] is over-estimated
    rc[a] = al[a]/lc[a];
  }
  if (sid == 0) {
    printf("lc = %d %d %d\n",lc[0],lc[1],lc[2]);
    printf("rc = %e %e %e\n",rc[0],rc[1],rc[2]);
  }

  /* Constants for potential truncation */
  rr = RCUT*RCUT; ri2 = 1.0/rr; ri6 = ri2*ri2*ri2; r1=sqrt(rr);
  Uc = 4.0*ri6*(ri6 - 1.0);
  Duc = -48.0*ri6*(ri6 - 0.5)/r1;
}

/*--------------------------------------------------------------------*/
void set_topology() {
/*--------------------------------------------------------------------
Defines a logical network topology.  Prepares a neighbor-node ID table,
nn, & a shift-vector table, sv, for internode message passing.  Also
prepares the node parity table, myparity.
--------------------------------------------------------------------*/
  /* Integer vectors to specify the six neighbor nodes */
  int iv[6][3] = {
    {-1,0,0}, {1,0,0}, {0,-1,0}, {0,1,0}, {0,0,-1}, {0,0,1}
  };
  int ku,a,k1[3];

  /* Set up neighbor tables, nn & sv */
  for (ku=0; ku<6; ku++) {
    /* Vector index of neighbor ku */
    for (a=0; a<3; a++)
      k1[a] = (vid[a]+iv[ku][a]+vproc[a])%vproc[a];
    /* Scalar neighbor ID, nn */
    nn[ku] = k1[0]*vproc[1]*vproc[2]+k1[1]*vproc[2]+k1[2];
    /* Shift vector, sv */
    for (a=0; a<3; a++) sv[ku][a] = al[a]*iv[ku][a];
  }

  /* Set up the node parity table, myparity */
  for (a=0; a<3; a++) {
    if (vproc[a] == 1)
      myparity[a] = 2;
  else if (vid[a]%2 == 0)
    myparity[a] = 0;
  else
    myparity[a] = 1;
 }
}

/*----------------------------------------------------------------*/
void init_conf() {
/*----------------------------------------------------------------
r are initialized to face-centered cubic (fcc) lattice positions.
rv are initialized with a random velocity corresponding to Temperature.
----------------------------------------------------------------*/
 double c[3],gap[3],e[3],vSum[3],gvSum[3],vMag;
 int j,a,nX,nY,nZ;
 double seed;
 /* FCC atoms in the original unit cell */
 double origAtom[4][3] = {{0.0, 0.0, 0.0}, {0.0, 0.5, 0.5},
                          {0.5, 0.0, 0.5}, {0.5, 0.5, 0.0}};

 /* Set up a face-centered cubic (fcc) lattice */
 for (a=0; a<3; a++) gap[a] = al[a]/InitUcell[a];
 n = 0;
 for (nZ=0; nZ<InitUcell[2]; nZ++) {
   c[2] = nZ*gap[2];
   for (nY=0; nY<InitUcell[1]; nY++) {
     c[1] = nY*gap[1];
     for (nX=0; nX<InitUcell[0]; nX++) {
       c[0] = nX*gap[0];
       for (j=0; j<4; j++) {
         for (a=0; a<3; a++)
           r[n][a] = c[a] + gap[a]*origAtom[j][a];
         ++n;
       }
     }
   }
 }
 /* Total # of atoms summed over processors */
 MPI_Allreduce(&n,&nglob,1,MPI_INT,MPI_SUM,MPI_COMM_WORLD);
 if (sid == 0) printf("nglob = %d\n",nglob);

 /* Generate random velocities */
 seed = 13597.0+sid;
 vMag = sqrt(3*InitTemp);
 for(a=0; a<3; a++) vSum[a] = 0.0;
 for(j=0; j<n; j++) {
   RandVec3(e,&seed);
   for (a=0; a<3; a++) {
     rv[j][a] = vMag*e[a];
     vSum[a] = vSum[a] + rv[j][a];
   }
 }
 MPI_Allreduce(vSum,gvSum,3,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);

 /* Make the total momentum zero */
 for (a=0; a<3; a++) gvSum[a] /= nglob;
 for (j=0; j<n; j++)
   for(a=0; a<3; a++) rv[j][a] -= gvSum[a];
}

/*--------------------------------------------------------------------*/
void single_step() {
/*--------------------------------------------------------------------
r & rv are propagated by DeltaT using the velocity-Verlet scheme.
--------------------------------------------------------------------*/
 int i,a;

 half_kick(); /* First half kick to obtain v(t+Dt/2) */
 for (i=0; i<n; i++) /* Update atomic coordinates to r(t+Dt) */
   for (a=0; a<3; a++) r[i][a] = r[i][a] + DeltaT*rv[i][a];
 atom_move();
 atom_copy();
 compute_accel(); /* Computes new accelerations, a(t+Dt) */
 half_kick(); /* Second half kick to obtain v(t+Dt) */
}

/*--------------------------------------------------------------------*/
void half_kick() {
/*--------------------------------------------------------------------
Accelerates atomic velocities, rv, by half the time step.
--------------------------------------------------------------------*/
 int i,a;
 for (i=0; i<n; i++)
   for (a=0; a<3; a++) rv[i][a] = rv[i][a]+DeltaTH*ra[i][a];
}

// FIRST CHANGE NEEDED
/*--------------------------------------------------------------------*/
```

```
void atom_copy() {
/*------------------------------------------------------------------------
Exchanges boundary-atom coordinates among neighbor nodes:  Makes
boundary-atom list, LSB, then sends & receives boundary atoms.
------------------------------------------------------------------------*/
 int kd,kdd,i,ku,inode,nsd,nrc,a;
 int nbnew = 0; /* # of "received" boundary atoms */
 double com1;

/* Main loop over x, y & z directions starts--------------------------*/
 // loop through x,y, or z direction
 for (kd=0; kd<3; kd++) {

  /* Make a boundary-atom list, LSB---------------------------------*/

  /* Reset the # of to-be-copied atoms for lower&higher directions */
  // lsb list of atoms to be sent to your neighbor
  // [2*kd+kdd]: idx of which atom to send  [0]: how many atoms to send to neighbor
  for (kdd=0; kdd<2; kdd++) lsb[2*kd+kdd][0] = 0;

  /* Scan all the residents & copies to identify boundary atoms */
  // n+nbnew: resident+cached atoms
  for (i=0; i<n+nbnew; i++) {
   for (kdd=0; kdd<2; kdd++) {
    ku = 2*kd+kdd; /* Neighbor ID */
    /* Add an atom to the boundary-atom list, LSB, for neighbor ku
       according to bit-condition function, bbd */
    if (bbd(r[i],ku)) lsb[ku][++(lsb[ku][0])] = i;
   }
  }

  /* Message passing------------------------------------------------*/

  com1=MPI_Wtime(); /* To calculate the communication time */

  /* Loop over the lower & higher directions */
  for (kdd=0; kdd<2; kdd++) {

   inode = nn[ku=2*kd+kdd]; /* Neighbor node ID */

   /* Send & receive the # of boundary atoms-----------------------*/

   nsd = lsb[ku][0]; /* # of atoms to be sent */
   // COMMENT BLOCK BELOW DUE TO Asynchronous
```

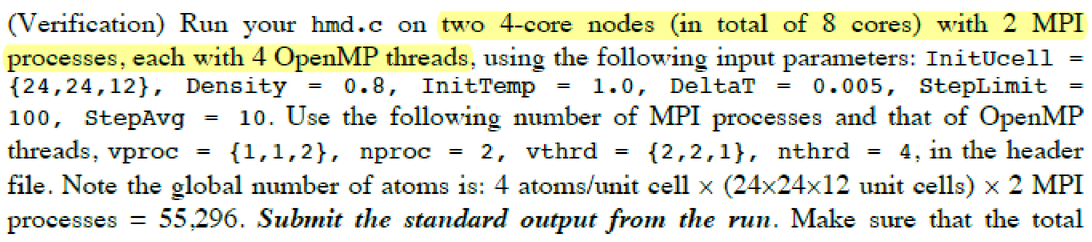
```
    /* Even node: send & recv */
    /* if (myparity[kd] == 0) {
      MPI_Send(&nsd,1,MPI_INT,inode,10,MPI_COMM_WORLD);
      MPI_Recv(&nrc,1,MPI_INT,MPI_ANY_SOURCE,10,
              MPI_COMM_WORLD,&status);
    }
    /* Odd node: recv & send */
    /* else if (myparity[kd] == 1) {
      MPI_Recv(&nrc,1,MPI_INT,MPI_ANY_SOURCE,10,
              MPI_COMM_WORLD,&status);
      MPI_Send(&nsd,1,MPI_INT,inode,10,MPI_COMM_WORLD);
    }
    /* Single layer: Exchange information with myself */
    /* else
      nrc = nsd;
    /* Now nrc is the # of atoms to be received */

    // NEW CODE FOR Asynchronous
    MPI_Irecv(&nrc,1,MPI_INT,MPI_ANY_SOURCE,10,
            MPI_COMM_WORLD,&request);
    MPI_Send(&nsd,1,MPI_INT,inode,10,MPI_COMM_WORLD);
    MPI_Wait(&request,&status); //handle to know which message to wait
    /* Send & receive information on boundary atoms-----------------*/

    // NEW CODE FOR Asynchronous, move this ahead of message buffering to optimize
    MPI_Irecv(dbufr,3*nrc,MPI_DOUBLE,MPI_ANY_SOURCE,20,
            MPI_COMM_WORLD,&request);

    /* Message buffering */
    // compose 1-d buffer file: move behind ireceive but before send will help optimize
computation time
    for (i=1; i<=nsd; i++)
      for (a=0; a<3; a++) /* Shift the coordinate origin */
        dbuf[3*(i-1)+a] = r[lsb[ku][i]][a]-sv[ku][a];

    // COMMENT OUT BLOCKS OF CODE BELOW
    /* Even node: send & recv */
    /*if (myparity[kd] == 0) {
      MPI_Send(dbuf,3*nsd,MPI_DOUBLE,inode,20,MPI_COMM_WORLD);
      MPI_Recv(dbufr,3*nrc,MPI_DOUBLE,MPI_ANY_SOURCE,20,
              MPI_COMM_WORLD,&status);
    }
    /* Odd node: recv & send */
    /*else if (myparity[kd] == 1) {
```

```
    MPI_Recv(dbufr,3*nrc,MPI_DOUBLE,MPI_ANY_SOURCE,20,
         MPI_COMM_WORLD,&status);
    MPI_Send(dbuf,3*nsd,MPI_DOUBLE,inode,20,MPI_COMM_WORLD);
   }
   /* Single layer: Exchange information with myself */
   /*else
    for (i=0; i<3*nrc; i++) dbufr[i] = dbuf[i];
   */

   // NEW CODE FOR Asynchronous
   MPI_Send(dbuf,3*nsd,MPI_DOUBLE,inode,20,MPI_COMM_WORLD);
   MPI_Wait(&request,&status); //handle to know which message to wait

   /* Message storing */
   for (i=0; i<nrc; i++)
    for (a=0; a<3; a++) r[n+nbnew+i][a] = dbufr[3*i+a];

   /* Increment the # of received boundary atoms */
   nbnew = nbnew+nrc;

   /* Internode synchronization */
   MPI_Barrier(MPI_COMM_WORLD);

  } /* Endfor lower & higher directions, kdd */

  comt += MPI_Wtime()-com1; /* Update communication time, COMT */

 } /* Endfor x, y & z directions, kd */

 /* Main loop over x, y & z directions ends--------------------------*/

 /* Update the # of received boundary atoms */
 nb = nbnew;
}

/*------------------------------------------------------------------*/
void compute_accel() {
/*-------------------------------------------------------------------
Given atomic coordinates, r[0:n+nb-1][], for the extended (i.e.,
resident & copied) system, computes the acceleration, ra[0:n-1][], for
the residents.
-------------------------------------------------------------------*/
 int i,j,a,lc2[3],lcyz2,lcxyz2,mc[3],c,mc1[3],c1;
 // int bintra; // ###
 double rrCut, lpe;
 // each array element stores the partial sum of the potential energy by a thread
 double lpe_td[nthrd]; // ###
 //double dr[3],rr,ri2,ri6,r1,rrCut,fcVal,f,vVal,lpe; // ###

 /* Reset the potential & forces */
 lpe = 0.0;
 // ###
 for (i=0; i<nthrd;i++) lpe_td[i] = 0.0;
 for (i=0; i<n; i++) for (a=0; a<3; a++) ra[i][a] = 0.0;

 /* Make a linked-cell list, lscl------------------------------------*/

 for (a=0; a<3; a++) lc2[a] = lc[a]+2;
 lcyz2 = lc2[1]*lc2[2];
 lcxyz2 = lc2[0]*lcyz2;

 /* Reset the headers, head */
 for (c=0; c<lcxyz2; c++) head[c] = EMPTY;

 /* Scan atoms to construct headers, head, & linked lists, lscl */

 for (i=0; i<n+nb; i++) {
   for (a=0; a<3; a++) mc[a] = (r[i][a]+rc[a])/rc[a];

   /* Translate the vector cell index, mc, to a scalar cell index */
   c = mc[0]*lcyz2+mc[1]*lc2[2]+mc[2];

   /* Link to the previous occupant (or EMPTY if you're the 1st) */
   lscl[i] = head[c];

   /* The last one goes to the header */
   head[c] = i;
 } /* Endfor atom i */

 /* Calculate pair interaction---------------------------------------*/

 rrCut = RCUT*RCUT;

 // ##### OpenMP parallel section
#pragma omp parallel private(a,mc,c,mc1,c1,i,j)
{
 // std is scalar thread index
 // vtd[3] x|y|z element of vector thread index
 // mofstp[3] x|y|z offset cell index of cell-block
 double dr[3],rr,ri2,ri6,r1,fcVal,f,vVal;
 int std, vtd[3],mofst[3];

 std = omp_get_thread_num();
 vtd[0] = std/(vthrd[1]*vthrd[2]);
 vtd[1] = (std/vthrd[2])%vthrd[1];
 vtd[2] = std%vthrd[2];
 for (a=0; a<3; a++)
    mofst[a] = vtd[a]*thbk[a];


 /* Scan inner cells */ // ###
 // for (mc[0]=1; mc[0]<=lc[0]; (mc[0])++)
 // for (mc[1]=1; mc[1]<=lc[1]; (mc[1])++)
 // for (mc[2]=1; mc[2]<=lc[2]; (mc[2])++) {
 for (mc[0]=mofst[0]+1; mc[0]<=mofst[0]+thbk[0]; (mc[0])++)
 for (mc[1]=mofst[1]+1; mc[1]<=mofst[1]+thbk[1]; (mc[1])++)
 for (mc[2]=mofst[2]+1; mc[2]<=mofst[2]+thbk[2]; (mc[2])++)  {
   /* Calculate a scalar cell index */
   c = mc[0]*lcyz2+mc[1]*lc2[2]+mc[2];
   /* Skip this cell if empty */
   if (head[c] == EMPTY) continue;

   /* Scan the neighbor cells (including itself) of cell c */
   for (mc1[0]=mc[0]-1; mc1[0]<=mc[0]+1; (mc1[0])++)
   for (mc1[1]=mc[1]-1; mc1[1]<=mc[1]+1; (mc1[1])++)
   for (mc1[2]=mc[2]-1; mc1[2]<=mc[2]+1; (mc1[2])++) {

     /* Calculate the scalar cell index of the neighbor cell */
     c1 = mc1[0]*lcyz2+mc1[1]*lc2[2]+mc1[2];
     /* Skip this neighbor cell if empty */
     if (head[c1] == EMPTY) continue;

     /* Scan atom i in cell c */
     i = head[c];
     while (i != EMPTY) {

       /* Scan atom j in cell c1 */
       j = head[c1];
       while (j != EMPTY) {

         /* No calculation with itself */
         if (j != i) {
           /* Logical flag: intra(true)- or inter(false)-pair atom */
           // bintra = (j < n); // ### remove race condition

           /* Pair vector dr = r[i] - r[j] */
           for (rr=0.0, a=0; a<3; a++) {
             dr[a] = r[i][a]-r[j][a];
             rr += dr[a]*dr[a];
           }

           /* Calculate potential & forces for intranode pairs (i < j)
              & all the internode pairs if rij < RCUT; note that for
              any copied atom, i < j */
           // if (i<j && rr<rrCut) { //###
           if (rr<rrCut) { // ### avoid critical sections
             ri2 = 1.0/rr; ri6 = ri2*ri2*ri2; r1 = sqrt(rr);
             fcVal = 48.0*ri2*ri6*(ri6-0.5) + Duc/r1;
             vVal = 4.0*ri6*(ri6-1.0) - Uc - Duc*(r1-RCUT);
             // if (bintra) lpe += vVal; else lpe += 0.5*vVal; // ###
             lpe_td[std] += 0.5*vVal; // ### count as half so that the other half is added later
 [i-j] is different from [j-i] interaction
           for (a=0; a<3; a++) {
               f = fcVal*dr[a];
               ra[i][a] += f;
               // if (bintra) ra[j][a] -= f; // ### mutually exclusive access to ra[][] for preventing
 race conditions
             }
           }
         } /* Endif not self */

         j = lscl[j];
       } /* Endwhile j not empty */

       i = lscl[i];
     } /* Endwhile i not empty */

   } /* Endfor neighbor cells, c1 */

  } /* Endfor central cell, c */

  }  /*End for parallel cell*/  // ###

   //Thread reduction  // ###
   for (i=0; i<nthrd; i++) lpe+= lpe_td[i];
   /* Global potential energy */
   MPI_Allreduce(&lpe,&potEnergy,1,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
 }

 /*--------------------------------------------------------------------*/
 void eval_props() {
 /*---------------------------------------------------------------------
 Evaluates physical properties: kinetic, potential & total energies.
 ---------------------------------------------------------------------*/
  double vv,lke;
  int i,a;

  /* Total kinetic energy */
  for (lke=0.0, i=0; i<n; i++) {
   for (vv=0.0, a=0; a<3; a++) vv += rv[i][a]*rv[i][a];
   lke += vv;
  }
  lke *= 0.5;
  MPI_Allreduce(&lke,&kinEnergy,1,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);

  /* Energy paer atom */
  kinEnergy /= nglob;
  potEnergy /= nglob;
  totEnergy = kinEnergy + potEnergy;
  temperature = kinEnergy*2.0/3.0;

  /* Print the computed properties */
  if (sid == 0) printf("%9.6f %9.6f %9.6f %9.6f\n",
               stepCount*DeltaT,temperature,potEnergy,totEnergy);
 }

 /*--------------------------------------------------------------------*/
 void atom_move() {
 /*---------------------------------------------------------------------
 Sends moved-out atoms to neighbor nodes and receives moved-in atoms
 from neighbor nodes.  Called with n, r[0:n-1] & rv[0:n-1], atom_move
 returns a new n' together with r[0:n'-1] & rv[0:n'-1].
 ---------------------------------------------------------------------*/

 /* Local variables------------------------------------------------------

 mvque[6][NBMAX]: mvque[ku][0] is the # of to-be-moved atoms to neighbor
  ku; MVQUE[ku][k>0] is the atom ID, used in r, of the k-th atom to be
  moved.
```

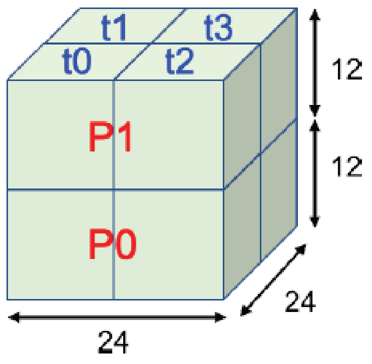
```
-------------------------------------------------------------------*/
 int mvque[6][NBMAX];
 int newim = 0; /* # of new immigrants */
 int ku,kd,i,kdd,kul,kuh,inode,ipt,a,nsd,nrc;
 double com1;

 /* Reset the # of to-be-moved atoms, MVQUE[][0] */
 for (ku=0; ku<6; ku++) mvque[ku][0] = 0;

 /* Main loop over x, y & z directions starts------------------------*/

 for (kd=0; kd<3; kd++) {

   /* Make a moved-atom list, mvque----------------------------------*/

   /* Scan all the residents & immigrants to list moved-out atoms */
   for (i=0; i<n+newim; i++) {
     kul = 2*kd  ; /* Neighbor ID */
     kuh = 2*kd+1;
     /* Register a to-be-copied atom in mvque[kul|kuh][] */
     if (r[i][0] > MOVED_OUT) { /* Don't scan moved-out atoms */
       /* Move to the lower direction */
       if (bmv(r[i],kul)) mvque[kul][++(mvque[kul][0])] = i;
       /* Move to the higher direction */
       else if (bmv(r[i],kuh)) mvque[kuh][++(mvque[kuh][0])] = i;
     }
   }

   /* Message passing with neighbor nodes----------------------------*/

   com1 = MPI_Wtime();

   /* Loop over the lower & higher directions------------------------*/

   for (kdd=0; kdd<2; kdd++) {

     inode = nn[ku=2*kd+kdd]; /* Neighbor node ID */

     /* Send atom-number information---------------------------------*/

     nsd = mvque[ku][0]; /* # of atoms to-be-sent */

     // COMMENT OUT
     /* Even node: send & recv */
```

```
    /*if (myparity[kd] == 0) {
      MPI_Send(&nsd,1,MPI_INT,inode,110,MPI_COMM_WORLD);
      MPI_Recv(&nrc,1,MPI_INT,MPI_ANY_SOURCE,110,
               MPI_COMM_WORLD,&status);
    }
    /* Odd node: recv & send */
    /*else if (myparity[kd] == 1) {
      MPI_Recv(&nrc,1,MPI_INT,MPI_ANY_SOURCE,110,
               MPI_COMM_WORLD,&status);
      MPI_Send(&nsd,1,MPI_INT,inode,110,MPI_COMM_WORLD);
    }
    /* Single layer: Exchange information with myself */
    /*else
      nrc = nsd;
    /* Now nrc is the # of atoms to be received */

    /* Send & receive information on boundary atoms-----------------*/

    // NEW CODE FOR Asynchronous
    MPI_Irecv(&nrc,1,MPI_INT,MPI_ANY_SOURCE,110,
             MPI_COMM_WORLD,&request);
    MPI_Send(&nsd,1,MPI_INT,inode,110,MPI_COMM_WORLD);
    MPI_Wait(&request,&status);



    /* Even node: send & recv, if not empty */
    /*if (myparity[kd] == 0) {
      MPI_Send(dbuf,6*nsd,MPI_DOUBLE,inode,120,MPI_COMM_WORLD);
      MPI_Recv(dbufr,6*nrc,MPI_DOUBLE,MPI_ANY_SOURCE,120,
               MPI_COMM_WORLD,&status);
    }
    /* Odd node: recv & send, if not empty */
    /*else if (myparity[kd] == 1) {
      MPI_Recv(dbufr,6*nrc,MPI_DOUBLE,MPI_ANY_SOURCE,120,
               MPI_COMM_WORLD,&status);
      MPI_Send(dbuf,6*nsd,MPI_DOUBLE,inode,120,MPI_COMM_WORLD);
    }
    /* Single layer: Exchange information with myself */
    /*else
      for (i=0; i<6*nrc; i++) dbufr[i] = dbuf[i];
    */

    // NEW CODE FOR Asynchronous
    MPI_Irecv(dbufr,6*nrc,MPI_DOUBLE,MPI_ANY_SOURCE,120,
          MPI_COMM_WORLD,&request);


     /* Message buffering */
     for (i=1; i<=nsd; i++)
       for (a=0; a<3; a++) {
         /* Shift the coordinate origin */
         dbuf[6*(i-1)  +a] = r [mvque[ku][i]][a]-sv[ku][a];
         dbuf[6*(i-1)+3+a] = rv[mvque[ku][i]][a];
         r[mvque[ku][i]][0] = MOVED_OUT; /* Mark the moved-out atom */
       }

    // NEW CODE FOR Asynchronous
    MPI_Send(dbuf,6*nsd,MPI_DOUBLE,inode,120,MPI_COMM_WORLD);
    MPI_Wait(&request,&status);

    /* Message storing */
    for (i=0; i<nrc; i++)
      for (a=0; a<3; a++) {
        r [n+newim+i][a] = dbufr[6*i  +a];
        rv[n+newim+i][a] = dbufr[6*i+3+a];
      }

    /* Increment the # of new immigrants */
    newim = newim+nrc;

    /* Internode synchronization */
    MPI_Barrier(MPI_COMM_WORLD);

   } /* Endfor lower & higher directions, kdd */

  comt=comt+MPI_Wtime()-com1;

 } /* Endfor x, y & z directions, kd */

 /* Main loop over x, y & z directions ends-------------------------*/

 /* Compress resident arrays including new immigrants */

 ipt = 0;
 for (i=0; i<n+newim; i++) {
   if (r[i][0] > MOVED_OUT) {
     for (a=0; a<3; a++) {
        r [ipt][a] = r [i][a];
        rv[ipt][a] = rv[i][a];
      }
      ++ipt;
    }
  }

  /* Update the compressed # of resident atoms */
  n = ipt;
}

/*------------------------------------------------------------------------
Bit condition functions:

1. bbd(ri,ku) is .true. if coordinate ri[3] is in the boundary to
     neighbor ku.
2. bmv(ri,ku) is .true. if an atom with coordinate ri[3] has moved out
     to neighbor ku.
------------------------------------------------------------------------*/
int bbd(double* ri, int ku) {
  int kd,kdd;
  kd = ku/2; /* x(0)|y(1)|z(2) direction */
  kdd = ku%2; /* Lower(0)|higher(1) direction */
  if (kdd == 0)
    return ri[kd] < RCUT;
  else
    return al[kd]-RCUT < ri[kd];
}
int bmv(double* ri, int ku) {
  int kd,kdd;
  kd = ku/2; /* x(0)|y(1)|z(2) direction */
  kdd = ku%2; /* Lower(0)|higher(1) direction */
  if (kdd == 0)
    return ri[kd] < 0.0;
  else
    return al[kd] < ri[kd];
}
```

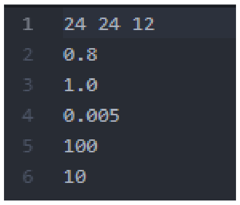
II. HMD verification



(Verification) Run your `hmd.c` on two 4-core nodes (in total of 8 cores) with 2 MPI processes, each with 4 OpenMP threads, using the following input parameters: `InitUcell = {24,24,12}, Density = 0.8, InitTemp = 1.0, DeltaT = 0.005, StepLimit = 100, StepAvg = 10`. Use the following number of MPI processes and that of OpenMP threads, `vproc = {1,1,2}, nproc = 2, vthrd = {2,2,1}, nthrd = 4`, in the header file. Note the global number of atoms is: 4 atoms/unit cell × (24×24×12 unit cells) × 2 MPI processes = 55,296. ***Submit the standard output from the run***. Make sure that the total



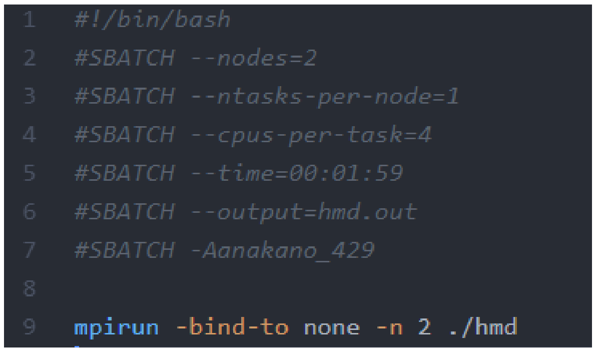
The input is given in hmd.h:



```
24 24 12
0.8
1.0
0.005
100
10
```

To complete this part, run the following on cluster:

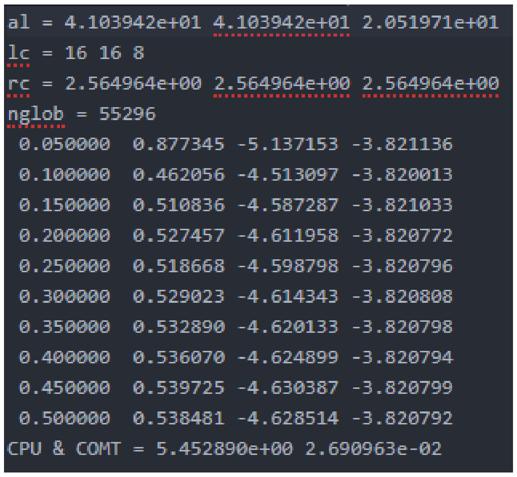
- Compile the program by: (The input file hmd.in and header hmd.h are automatically read by the following scripts in hmd.c)
  - **Mpicc -O -o hmd hmd.c -lm -fopenmp**
- Run sbatch:
  - **Sbatch hmd.sl**



```
#!/bin/bash
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=4
#SBATCH --time=00:01:59
#SBATCH --output=hmd.out
#SBATCH -Aanakano_429

mpirun -bind-to none -n 2 ./hmd
```

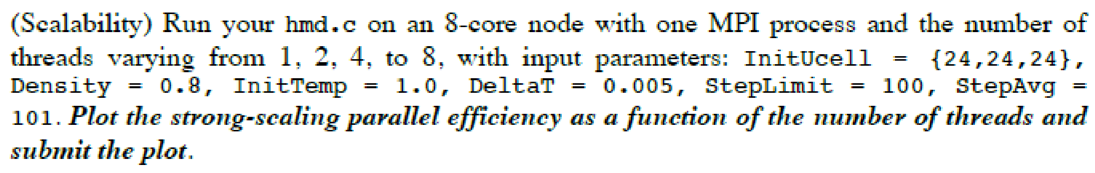
- This sbatch will get 2 computing nodes, each with 4 processors/cores. The job consists of 2 processes/tasks (regulated by MPI to assign each proecess to each node). Each process/task consists of 4 threads, each of which is assigned to each core utilizing OpenMP framework. Then it will generate output hmd.out.

Below is the output hmd.out. The total energy is the same to the given file in the question prompt.



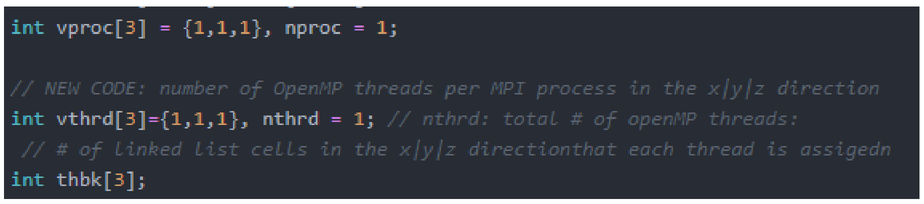
```
al = 4.103942e+01 4.103942e+01 2.051971e+01
lc = 16 16 8
rc = 2.564964e+00 2.564964e+00 2.564964e+00
nglob = 55296
 0.050000  0.877345 -5.137153 -3.821136
 0.100000  0.462056 -4.513097 -3.820013
 0.150000  0.510836 -4.587287 -3.821033
 0.200000  0.527457 -4.611958 -3.820772
 0.250000  0.518668 -4.598798 -3.820796
 0.300000  0.529023 -4.614343 -3.820808
 0.350000  0.532890 -4.620133 -3.820798
 0.400000  0.536070 -4.624899 -3.820794
 0.450000  0.539725 -4.630387 -3.820799
 0.500000  0.538481 -4.628514 -3.820792
CPU & COMT = 5.452890e+00 2.690963e-02
```

III. HMD scalability



(Scalability) Run your `hmd.c` on an 8-core node with one MPI process and the number of threads varying from 1, 2, 4, to 8, with input parameters: `InitUcell = {24,24,24}, Density = 0.8, InitTemp = 1.0, DeltaT = 0.005, StepLimit = 100, StepAvg = 101`. ***Plot the strong-scaling parallel efficiency as a function of the number of threads and submit the plot.***

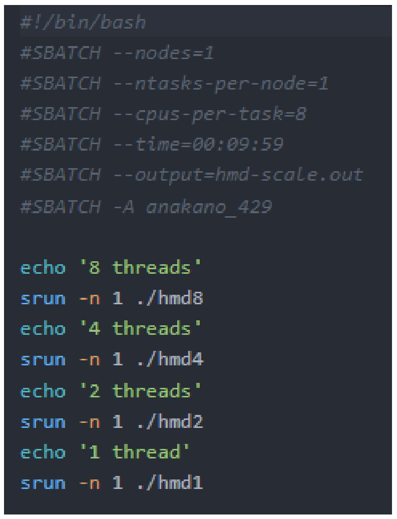
Based on the original hmd.h and hmd.c, I need to create a set of new header files with different specifications given by the requirements. For example, the hmd1.h is modified as below so that number of threads to be used is 1.



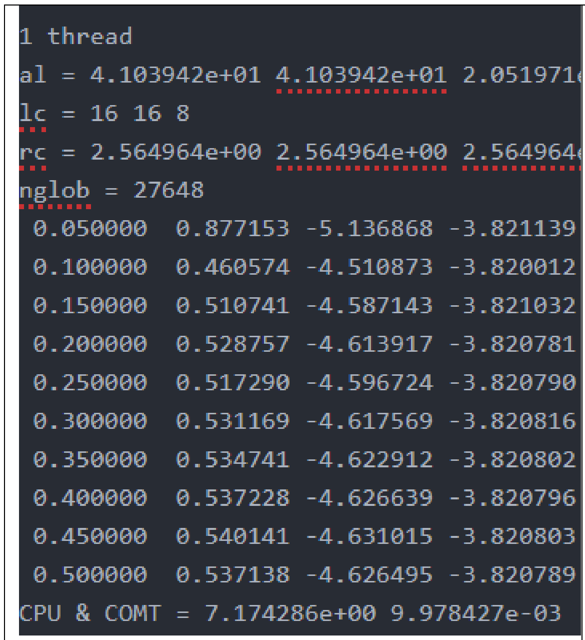
```
int vproc[3] = {1,1,1}, nproc = 1;

// NEW CODE: number of OpenMP threads per MPI process in the x|y|z direction
int vthrd[3]={1,1,1}, nthrd = 1; // nthrd: total # of openMP threads:
 // # of linked list cells in the x|y|z directionthat each thread is assigedn
int thbk[3];
```

- For hmd2: **int vthrd[3]={2,1,1}, nthrd = 2;**
- For hmd4: **int vthrd[3]={2,2,1}, nthrd = 4;**
- For hmd8: **int vthrd[3]={2,2,2}, nthrd = 8;**

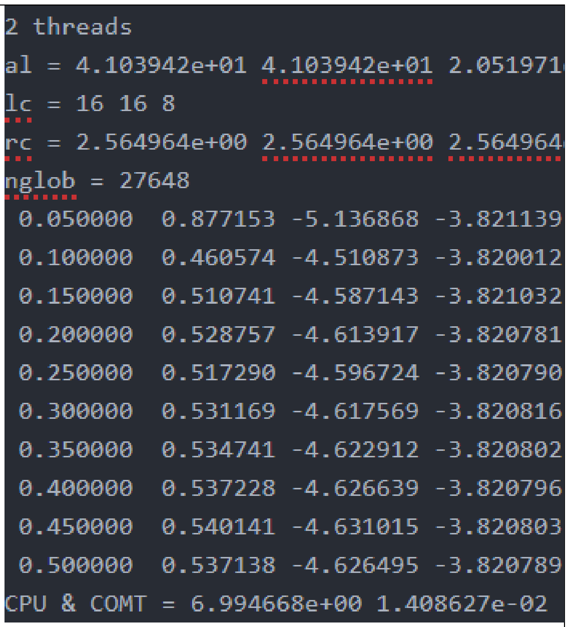
The file name will be saved as given in the slurm file: hmd1, hmd2, hmd4, hmd8. This slum file requests for 1 computing node to handle 1 task, of which 8 cores are available in this node.



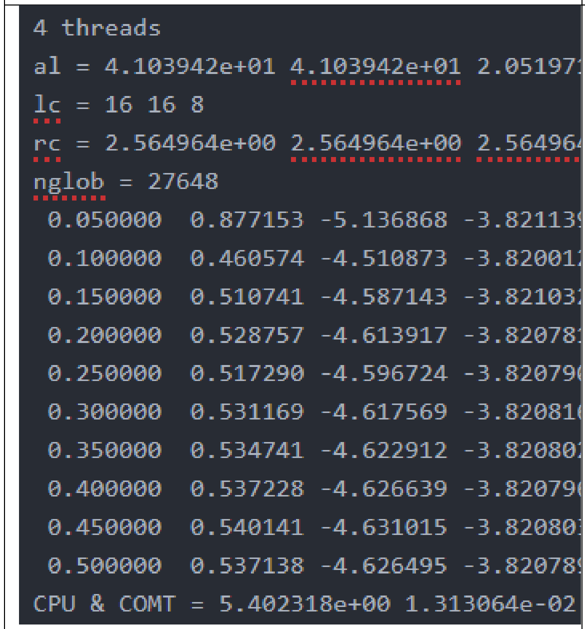
```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=8
#SBATCH --time=00:09:59
#SBATCH --output=hmd-scale.out
#SBATCH -A anakano_429

echo '8 threads'
srun -n 1 ./hmd8
echo '4 threads'
srun -n 1 ./hmd4
echo '2 threads'
srun -n 1 ./hmd2
echo '1 thread'
srun -n 1 ./hmd1
```



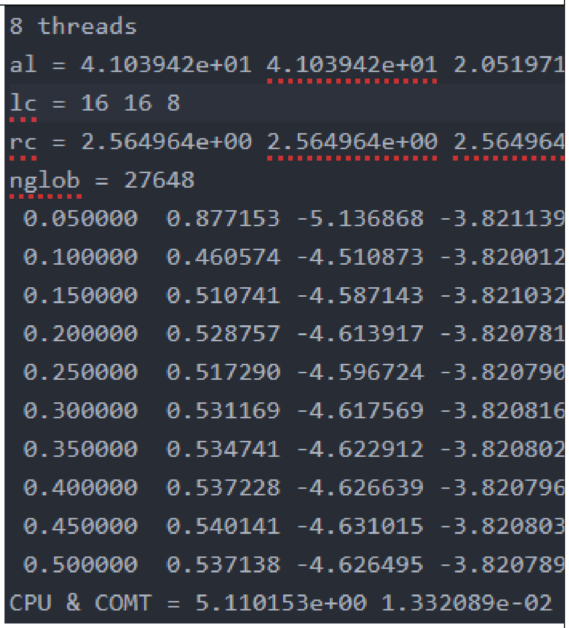
```
1 thread
al = 4.103942e+01 4.103942e+01 2.051971
lc = 16 16 8
rc = 2.564964e+00 2.564964e+00 2.564964
nglob = 27648
 0.050000  0.877153 -5.136868 -3.821139
 0.100000  0.460574 -4.510873 -3.820012
 0.150000  0.510741 -4.587143 -3.821032
 0.200000  0.528757 -4.613917 -3.820781
 0.250000  0.517290 -4.596724 -3.820790
 0.300000  0.531169 -4.617569 -3.820816
 0.350000  0.534741 -4.622912 -3.820802
 0.400000  0.537228 -4.626639 -3.820796
 0.450000  0.540141 -4.631015 -3.820803
 0.500000  0.537138 -4.626495 -3.820789
CPU & COMT = 7.174286e+00 9.978427e-03
```



```
2 threads
al = 4.103942e+01 4.103942e+01 2.051971
lc = 16 16 8
rc = 2.564964e+00 2.564964e+00 2.564964
nglob = 27648
 0.050000  0.877153 -5.136868 -3.821139
 0.100000  0.460574 -4.510873 -3.820012
 0.150000  0.510741 -4.587143 -3.821032
 0.200000  0.528757 -4.613917 -3.820781
 0.250000  0.517290 -4.596724 -3.820790
 0.300000  0.531169 -4.617569 -3.820816
 0.350000  0.534741 -4.622912 -3.820802
 0.400000  0.537228 -4.626639 -3.820796
 0.450000  0.540141 -4.631015 -3.820803
 0.500000  0.537138 -4.626495 -3.820789
CPU & COMT = 6.994668e+00 1.408627e-02
```



```
4 threads
al = 4.103942e+01 4.103942e+01 2.05197
lc = 16 16 8
rc = 2.564964e+00 2.564964e+00 2.56496
nglob = 27648
 0.050000  0.877153 -5.136868 -3.82113
 0.100000  0.460574 -4.510873 -3.82001
 0.150000  0.510741 -4.587143 -3.82103
 0.200000  0.528757 -4.613917 -3.82078
 0.250000  0.517290 -4.596724 -3.82079
 0.300000  0.531169 -4.617569 -3.82081
 0.350000  0.534741 -4.622912 -3.82080
 0.400000  0.537228 -4.626639 -3.82079
 0.450000  0.540141 -4.631015 -3.82080
 0.500000  0.537138 -4.626495 -3.82078
CPU & COMT = 5.402318e+00 1.313064e-02
```

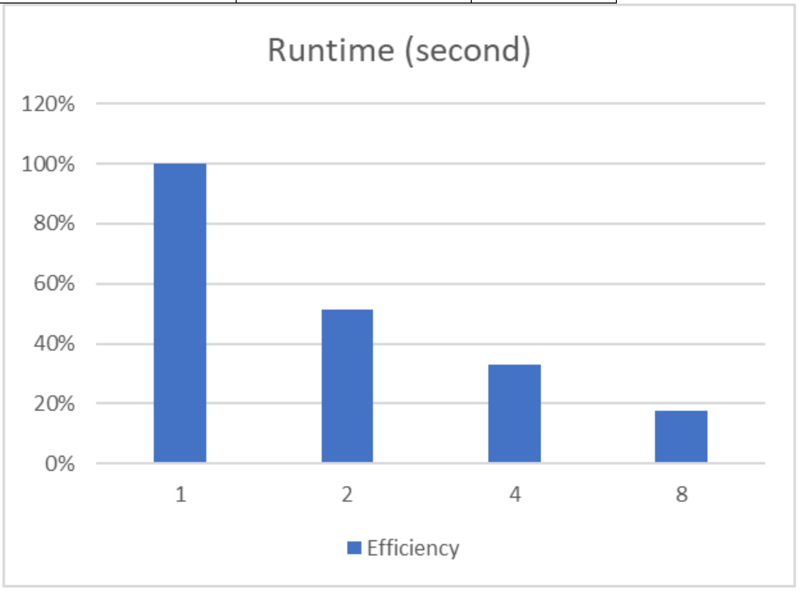


```
8 threads
al = 4.103942e+01 4.103942e+01 2.051971
lc = 16 16 8
rc = 2.564964e+00 2.564964e+00 2.564964
nglob = 27648
 0.050000  0.877153 -5.136868 -3.821139
 0.100000  0.460574 -4.510873 -3.820012
 0.150000  0.510741 -4.587143 -3.821032
 0.200000  0.528757 -4.613917 -3.820781
 0.250000  0.517290 -4.596724 -3.820790
 0.300000  0.531169 -4.617569 -3.820816
 0.350000  0.534741 -4.622912 -3.820802
 0.400000  0.537228 -4.626639 -3.820796
 0.450000  0.540141 -4.631015 -3.820803
 0.500000  0.537138 -4.626495 -3.820789
CPU & COMT = 5.110153e+00 1.332089e-02
```

The computing time output is summarized in the table and graph below:

Number of threads	CPU Running time	Efficiency
1	7.174286	100%
2	6.994668	51%
4	5.402318	33%

8	5.110153	18%


Runtime (second)
Efficiency