Minh Tran

Assignment 5 CSCI 596: Parallel Molecular Dynamics

1. 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)

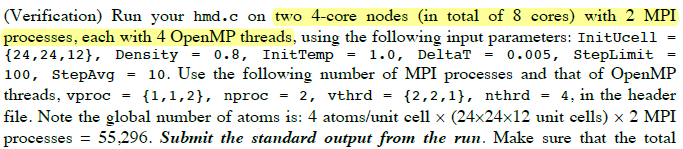
return ri[kd] < 0.0;

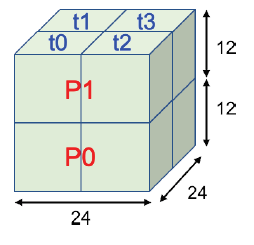
else

return al[kd] < ri[kd];

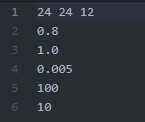
}

1. HMD verification



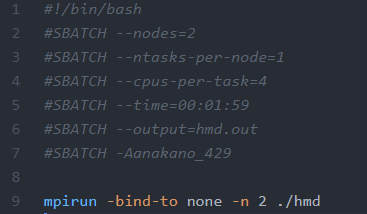


The input is given in hmd.h:



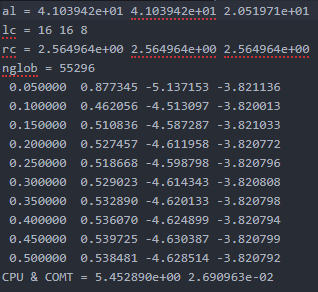
To complete this part, run the following on cluster:

* 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**

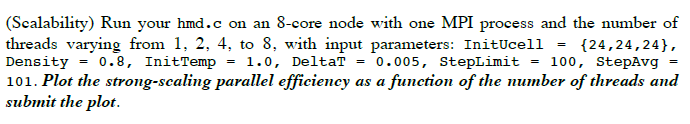


* + This sbatch will get 2 computing nodes, each with 4 processors/cores. The job consists of 2 processes/tasks (regulated by MPI to assign each proecess to each node). Each process/task consists of 4 threads, each of which is assigned to each core utilizing OpenMP framework. Then it will generate output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

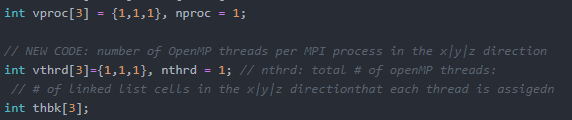
Below is the output hmd.out. The total energy is the same to the given file in the question prompt.



1. HMD scalability

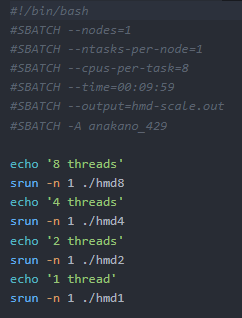


Based on the original hmd.h and hmd.c, I need to create a set of new header files with different specifications given by the requirements. For example, the hmd1.h is modified as below so that number of threads to be used is 1.



* 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.



|  |  |
| --- | --- |
|  |  |
|  |  |

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% |

