Assignment 4- Parallel Molecular Dynamics
CSCI 596: Scientific Computing & Visualization

Anup V Kanale

October 18, 2017

This purpose of this assignment is run molecular dynamics on multiple processors to understand asynchronous message passing and "in-situ" data analysis.

1 Asynchronous Messages

In this section, we use `MPI_Irecv()` and `MPI_Send()` with an `MPI_Wait()` in the program `pmd.c`. The asynchronous messages make the deadlock-avoidance scheme unnecessary, and thus there is no need to use different orders of send and receive calls for even and odd-parity processes.

The new asynchronous implementation takes $5.13e-01$ seconds, whereas the original implementation takes $6.28e-01$ seconds. This observed speedup is $\approx 20\%$. The major modifications were made in the functions `atom_copy` and `atom_move`. To keep this report short, only these parts are shown in Appendix A.1.

2 Communicators

In this section, we basically allocate half the processors for the computations, and the other half for analysing the data from the computations. Specifically, following the lecture note on "In situ analysis of molecular dynamics simulation data using communicators", `pmd.c` is modified such that as many number of processes as that for MD simulations are spawned to calculate the probability density function (PDF) for the atomic velocity.

The plot of the probability density function is as shown below: The major modification was in

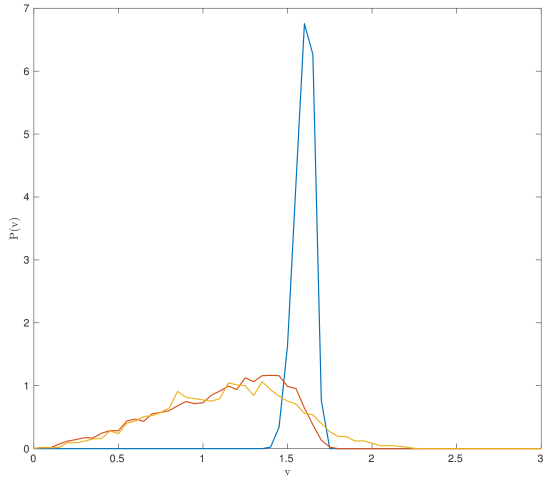

Figure 1: Probability density function

the `main` function, `calc_pv()` was provided. Only the modified portion is shown in appendix A.2 to keep this report short.

A Appendix

A.1 Task I

```
/*------------------------------------------------------------------------*/
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--------------------------*/

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

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

    /* Reset the # of to-be-copied atoms for lower&higher directions */
    for (kdd=0; kdd<2; kdd++) lsb[2*kd+kdd][0] = 0;

    /* Scan all the residents & copies to identify boundary 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 */

//---START MODIFIED----------------------------------------------------------------------------------

    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);

    /* Send & receive information on boundary atoms--------------------*/
    MPI_Irecv(dbufr,3*nrc,MPI_DOUBLE,MPI_ANY_SOURCE,20,MPI_COMM_WORLD,&Request);

    /* Message buffering */
    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];

    MPI_Send(dbuf,3*nsd,MPI_DOUBLE,inode,20,MPI_COMM_WORLD);
    MPI_Wait(&Request, &status);
//---END MODIFIED----------------------------------------------------------------------------------


    /* 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 */
```

```c
  nb = nbnew;
}


/*--------------------------------------------------------------------*/
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 */

  //---START MODIFIED---------------------------------------------------
      /* Send & receive information on boundary atoms-----------------*/
      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);

      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 */
        }

      MPI_Send(dbuf,6*nsd,MPI_DOUBLE,inode,120,MPI_COMM_WORLD);
      MPI_Wait(&Request, &status);
  //---END MODIFIED-----------------------------------------------------


      /* 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];
}
```

pmd_async.c

A.2 Task II

```
/*----------------------------------------------------------------------
Program pmd.c performs parallel molecular-dynamics for Lennard-Jones
systems using the Message Passing Interface (MPI) standard.
----------------------------------------------------------------------*/
#include "pmd_async_task2.h"
#define VMAX 5.0  // Max. velocity value to construct a velocity
    histogram
#define NBIN 100  // # of bins in the histogram

FILE *fpv;

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

  MPI_Init(&argc,&argv); /* Initialize the MPI environment */
  //MPI_Comm_rank(MPI_COMM_WORLD, &sid);  /* My processor ID */

  MPI_Comm_rank(MPI_COMM_WORLD, &gid); //Global rank
  md_shit = gid%2; // = 1 (MD workers) or 0 (analysis workers)
  MPI_Comm_split(MPI_COMM_WORLD,md_shit,0,&workers);
  MPI_Comm_rank(workers,&sid); // Rank in workers

  /* Vector index of this processor */
  vid[0] = sid/(vproc[1]*vproc[2]);
  vid[1] = (sid/vproc[2])%vproc[1];
  vid[2] = sid%vproc[2];

  init_params();
  if (md_shit) {
    set_topology();
    init_conf();
    atom_copy();
    compute_accel();
  }
  else
    if (sid == 0)
     fpv = fopen("pv.dat","w");

  //printf("%d\n", md_shit);
  cpu1 = MPI_Wtime();
  for (stepCount=1; stepCount<=StepLimit; stepCount++) {
  if (md_shit){
    printf("%d\n", md_shit);
    single_step();
  }
  if (stepCount%StepAvg == 0) {
    if (md_shit) {
        // Send # of atoms, n, to rank gid-1 in MPI_COMM_WORLD
        MPI_Send(&n, 1, MPI_INT, gid-1, 1000, MPI_COMM_WORLD);
        // Compose message to be sent
        for(i=0; i<n; i++)
            for(a=0; a<3; a++)
                dbuf[3*i+a] = rv[i][a];
        // Send velocities of n atoms to rank gid-1 in MPI_COMM_WORLD
        MPI_Send(dbuf, 3*n, MPI_DOUBLE, gid-1, 2000, MPI_COMM_WORLD);
        eval_props();
    }
    else {
        // Receive # of atoms, n, from rank gid+1 in MPI_COMM_WORLD
        MPI_Recv(&n, 1, MPI_INT, gid+1, 1000, MPI_COMM_WORLD,&status);
        // Receive velocities of n atoms from rank gid+1 in MPI_COMM_WORLD
        MPI_Recv(dbufr, 3*n, MPI_DOUBLE, gid+1, 2000, MPI_COMM_WORLD,&status);
        for(i=0; i<n; i++)
            for(a=0; a<3; a++)
                rv[i][a] = dbufr[3*i+a];
        calc_pv();
    } // end if
  } // end if
  } // end for


  cpu = MPI_Wtime() - cpu1;
  if(md_shit && sid==0)
    printf("CPU & COMT = %le %le\n",cpu,comt);
  if(!md_shit && sid==0)
    fclose(fpv);

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

pmd_async_task2.c