Assignment 5- Hybrid MPI + MP
CSCI 596: Scientific Computing & Visualization

Anup V Kanale

October 18, 2017

The goal of this assignment is to make use of both MPI and OpenMP in the supplied parallel molecular dynamics code. MPI is used to pass messages between processors, whereas OpenMP uses cores within a processor.

1 Task I– hmd.c

The modifications were made mainly in compute-accel() function (apart from `init_params()`) and the main(). `compute_accel` and the `main` function are attached in appendix. Apart from this, minor modifications were made in the header file, mainly to declare the number of threads. The rest of the code, which already has MPI was retained from `pmd.c`.

2 Task II– Verification

To verify, the program `hmd.c` was run on HPC. The results (energy) match the parallel molecular dynamics code (`pmd.c`). The output of the run is shown below.

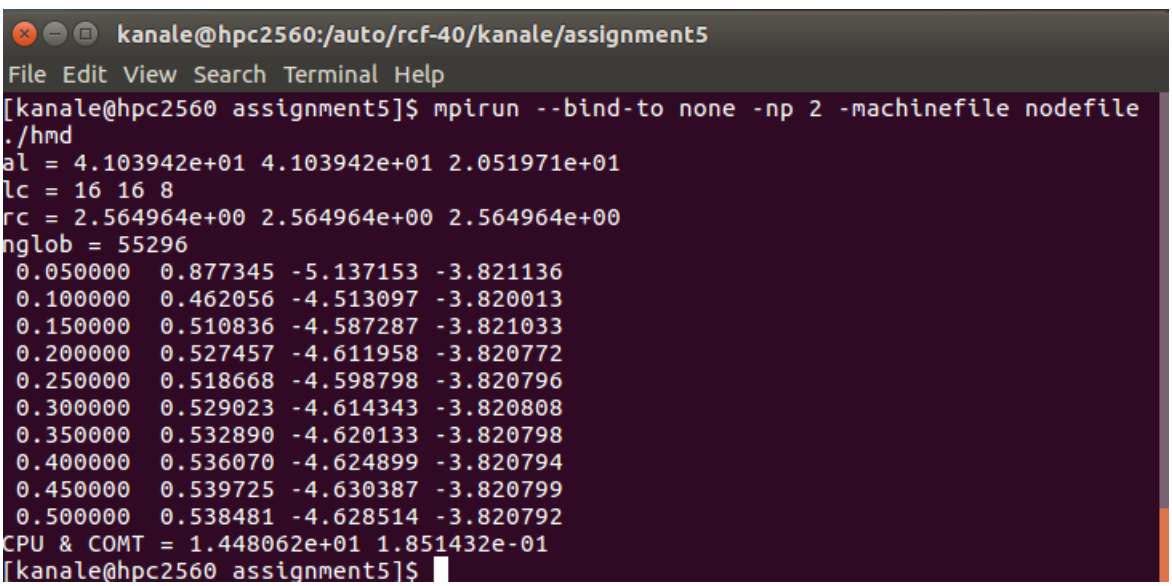

Figure 1: Output for task 2– Verification

3 Task III– Scalability

In this task, the scalability of the program is tested w.r.t the number of threads. The plot is attached below.

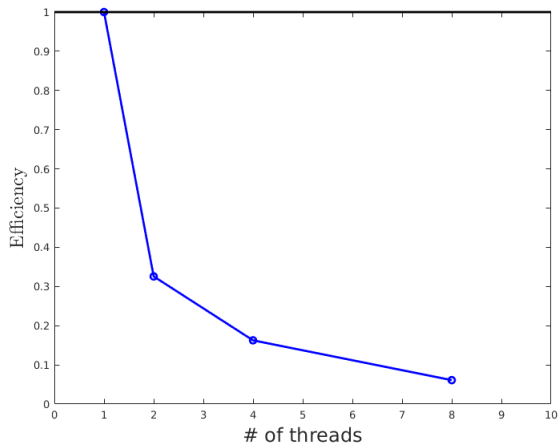

Figure 2: Output for task 3– Scaling

A Appendix– `hmd.c`

```
/*-----------------------------------------------------------------------
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); // for the pragma stuff

  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 */
  for (a=0; a<3; a++) {
    lc[a] = al[a]/RCUT; /* Cell size âĽě potential cutoff */
    /* Size of cell block that each thread is assigned */
    thbk[a] = lc[a]/vthrd[a];
    /* # of cells = integer multiple of the # of threads */
    lc[a] = thbk[a]*vthrd[a]; /* Adjust # of cells/MPI process */
    rc[a] = al[a]/lc[a]; /* Linked-list cell length */
  }
  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;

//  printf("finished init params");
}


/*-----------------------------------------------------------------*/
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;
  double  lpe_td[nthrd];

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

  #pragma omp parallel private(mc,a,c,c1,mc1,i,j,)
  {

  int std,vtd[3],mofst[3];
  double dr[3],rr,ri2,ri6,r1,fcVal,f,vVal;

  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]=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);

            /* 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 (rr<rrCut) {
              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;
              for (a=0; a<3; a++) {
                f = fcVal*dr[a];
                ra[i][a] += f;
                //if (bintra) ra[j][a] -= f;
              }
            }
          } /* 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 */

  } // Ending pragma or omp parallel
  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);
}
```

hmd.c