Parallelizing a Molecular Dynamics Code Using IPT

1. Molecular Dynamics (MD) Simulation Application

The MD simulation application is used in multiple domains (*e.g.*, biochemistry, biophysics, and material science) for studying the movement of atoms and molecules. Typically, the simulation is run for a fixed period of time during which the atoms and molecules are allowed to interact with each other [1]. During the interaction, the atoms and molecules exert force on each other, in a constrained or an unconstrained manner, thus giving a dynamic view of their interaction over a period of time.

The MD simulation application that we are considering here is designed to follow the path of particles that exert force on each other and are not constrained by any walls [2]. Therefore, after colliding, the particles move past each other. This MD code uses the velocity Verlet time integration scheme and the particles in the simulation interact with a central pair potential [2].

The compute-intensive steps in this application are related to calculating the force and energies (potential energy and kinetic energy) in each time-step, as well as updating the values of the positions, velocities, and accelerations of the particles (*i.e.*, the atoms and molecules) in the simulation. The size of the system (*i.e.,* the number of particles in the simulation), the length of time between the evaluation of energies and force (*i.e.*, the size of the integration step), and the time duration for which the interaction of particles needs to be studied impact the overall runtime of the simulation. In real-world scenarios, often there are several thousand particles in an MD simulation and studying their movement using a serial implementation of the code can take a large amount of time. Therefore, parallel computing is used to reduce the overall run-time of such simulations.

2. Serial Version of the MD Code and its Parallelization

Let us consider the serial version of the code for the MD simulation application. The complete code for this application is available at [2]. After profiling the code with `gprof`, we can see that the function named `compute` is the most time-consuming function and is a good candidate for parallelization. The program spends about 98.9% of its time in this function. A code snippet of the `compute` function is shown in Figure 1.

```
1.  for ( k = 0; k < np; k++ ){
2.      //Compute the potential energy (pe) and forces (f).
3.      for ( i = 0; i < nd; i++ ){
4.        f[i+k*nd] = 0.0;
5.      }
6.      for ( j = 0; j < np; j++ ){
7.        if ( k != j ){
8.           d = dist ( nd, pos+k*nd, pos+j*nd, rij );
9.           if ( d < PI2 ) {
10.            d2 = d;
11.          } else{
12.            d2 = PI2;}
13.          pe = pe + 0.5 * pow ( sin ( d2 ), 2 );
14.          for ( i = 0; i < nd; i++ ){
15.            f[i+k*nd] = f[i+k*nd] - rij[i] * sin ( 2.0 * d2 ) / d; }
16.       }
17.      }
18.      //Compute the kinetic energy (ke).
19.      for ( i = 0; i < nd; i++ ) {
20.          ke = ke + vel[i+k*nd] * vel[i+k*nd]; }
21.  }
```

Figure 1: Snippet of the `compute` function – serial version of the MD code

The number of iterations of the for-loop beginning at line # 1 of Figure 1 is equal to the number of particles (`np`) in the simulation. The potential energy (`pe`), force (`f`), and kinetic energy (`ke`) of each particle are calculated in each iteration, and their values across all the iterations are added together. The number of spatial dimensions for the simulation is represented by `nd` and the distance between the particles is represented by `d` in the code snippet shown in Figure 1. The values of the elements in the displacement vector `rij` are calculated for each particle in the function named `dist` in each iteration of the for-loop beginning at line # 6 of the for-loop. The variable `PI2` shown in the code snippet in Figure 1 represents the value of the constant `pi` divided by 2, that is, (3.14/2). Except the loop-variables that are of type `integer`, all other values in this example are of type `double`.

There are no dependencies between the iterations of the for-loop at line # 1 of the code in Figure 1. Hence, this for-loop can be parallelized by distributing the computations in the loop across multiple threads or processes. Once all the threads or processes have finished their share of computations and have their local results ready, we can combine those results in a meaningful manner to obtain a global result. This global result should match the results of running the application in the serial mode as closely as possible (some rounding off errors may be permissible). Combining the locally computed values of variables while applying a mathematical operation (***e.g.***, sum or multiplication) is referred to as **reduction**. For combining the values of small local arrays into a large global array when the local arrays are computed using processes having separate address space - as is the case in MPI - a **gather** operation is used. Both **reduction** and **gather** operations are classified under the pattern named "**data collection"** in IPT.

In the MD example, if the for-loop at line # 1 of Figure 1 is parallelized, the values of the variables `pe` and `ke` will be computed by each thread or process participating in the computation for a certain number of iterations. However, these values should be collected together and added using reduction to obtain the same result as one would get without parallelization.

Each thread or MPI process when working in parallel updates certain number of elements of the array `f`. If the threads are used for parallel computation, this array can be in a

shared memory region. Each thread can work on updating the values of certain number of elements in the same array. Hence, no extra step to combine the updated elements of the array is needed. However, we do need to use a gather operation to combine the updates made to the elements of the array when MPI processes are used because the MPI processes have independent address space.

3. Using IPT

IPT is invoked from the command-line as shown below and it expects the path to the input source code (`md.c`) as its argument:

```
$ IPT md.c
```

After analyzing the input source code (here `md.c`), IPT poses some high-level questions to the user. To begin with, IPT prompts the user to select a parallel programming paradigm. The exact sequence of the questions posed by IPT for parallelizing the MD application is shown in the next three sections. The input file remains unmodified and IPT generates new output files containing the parallel versions of the code that is provided as input. As explained in Section 2, for the MD application, the `compute` function is a good candidate for parallelization. The next three sections demonstrate the steps for using IPT to parallelize the code shown in Figure 1.

4. Generating the OpenMP version of the MD Code using IPT

The steps for generating the OpenMP version of the MD code using IPT are shown in Figure 2. The questions posed by IPT during the process of generating the OpenMP version of the MD application are shown in boldface in Figure 2. The details related to the meaning of the questions are available in the IPT documentation.

```
$ IPT md.c

NOTE: We currently support only C and C++ programs.

Please select a parallel programming model from the following available options:
1. MPI
2. OpenMP
3. CUDA
2

Would you like to parallelize a for-loop?(Y/N)
y

Please choose the function that you want to parallelize from the list below
1 : main
2 : compute
3 : cpu_time
4 : dist
5 : initialize
6 : r8mat_uniform_ab
7 : timestamp
8 : update
2

Note: With your response, you will be selecting or declining the parallelization of the
outermost for-loop in the code region shown below. If instead of the outermost for-loop,
there are any inner for-loops in this code region that you are interested in
parallelizing, then, you will be able to select those at a later stage.

for (k = 0; k < np; k++) {
  //Compute the potential energy and forces.
  for (i = 0; i < nd; i++) {
    f[i + (k * nd)] = 0.0;
  }
```

OpenMP is selected

The `compute` function is selected

The loop shown in Figure 1 is found by IPT

```
  for (j = 0; j < np; j++) {
    if (k != j) {
      d = dist(nd,(pos + (k * nd)),(pos + (j * nd)),rij);
      //Attribute half of the potential energy to particle J.
      if (d < PI2) {
        d2 = d;
      }
      else {
        d2 = PI2;
      }
      pe = (pe + (0.5 * pow(sin(d2),2)));
      for (i = 0; i < nd; i++) {
        f[i + (k * nd)] = (f[i + (k * nd)] - ((rij[i] * sin((2.0 * d2))) / d));
      }
    }
  }
  //Compute the kinetic energy.
  for (i = 0; i < nd; i++) {
    ke = (ke + (vel[i + (k * nd)] * vel[i + (k * nd)]));
  }
}
```

Is this the for loop you are looking for?(y/n)
y

Do you want to perform reduction on any variable?(Y/N)
y

Reduction variables are the variables that should be updated by the OpenMP threads and then acumulated according to a mathematical operation like sum, multiplication, etc.

Please select a variable to perform the reduce operation on (format 1,2,3,4 etc.). List of possible variables are:
1. nd type is int
2. k type is int
3. np type is int
4. d type is double
5. PI2 type is double
6. d2 type is double
7. pe type is double
8. ke type is double
7,8

`ke` and `pe` are selected for reduction

Please enter the type of reduction you wish for variable [pe]
1. Addition
2. Subtraction
3. Min
4. Max
5. Multiplication
1

Please enter the type of reduction you wish for variable [ke]
1. Addition
2. Subtraction
3. Min
4. Max
5. Multiplication
1

IPT is unable to perform the dependency analysis of the array named [f] in the region of code that you wish to parallelize. Please enter 1 if the entire array is being updated in a single iteration of the loop that you selected for parallelization, or, enter 2 otherwise.
2

IPT is unable to perform the dependency analysis of the array named [rij] in the region of code that you wish to parallelize. Please enter 1 if the entire array is being updated in a single iteration of the loop that you selected for parallelization, or, enter 2 otherwise.
1

```
Are there any lines of code that you would like to run either using a single thread at a
time (hence, one thread after another), or using only one thread?(Y/N)
n

Would you like to parallelize another loop?(Y/N)
n

Are you writing/printing anything from the parallelized region of the code?(Y/N)
n

Running Consistency Tests
```

Figure 2: Steps for generating the OpenMP version of the MD code using IPT

After the user follows the steps shown in Figure 2, IPT generates an output file named **rose_md_OpenMP.c** in the current working directory. A snippet of the OpenMP code generated by IPT is shown in Figure 3 and the code inserted by IPT is highlighted in boldface. You can compare this code snippet with the code snippet shown in Figure 1 to learn about the steps required for parallelizing the computations in a for-loop. The complete code is provided in the appendix. As can be noticed from the OpenMP version of the code in the appendix, IPT inserted the statement for including the OpenMP header file. It created the directives for starting a team of parallel threads for executing the for-loop and performing the reduction operation. On the basis of its analysis of the input code, and the specifications provided by the user, IPT creates the clauses of the OpenMP directives. The meaning and purpose of the OpenMP clauses are explained in the IPT documentation.

```
#pragma omp parallel default(none) shared(pe,ke,np,f,pos,vel,nd,PI2) private(k,i,j,d,d2)
firstprivate(rij)
{

#pragma omp for reduction ( + :pe,ke)
  for (k = 0; k < np; k++) {
  //Compute the potential energy and forces.
    for (i = 0; i < nd; i++) {
      f[i + (k * nd)] = 0.0;
    }
    for (j = 0; j < np; j++) {
      if (k != j) {
        d = dist(nd,(pos + (k * nd)),(pos + (j * nd)),rij);
        //Attribute half of the potential energy to particle J.
        if (d < PI2) {
          d2 = d;
        }
        else {
          d2 = PI2;
        }
        pe = (pe + (0.5 * pow(sin(d2),2)));
        for (i = 0; i < nd; i++) {
          f[i + (k * nd)] = (f[i + (k * nd)] - ((rij[i] * sin((2.0 * d2))) / d));
        }
      }
    }
    //Compute the kinetic energy.
    for (i = 0; i < nd; i++) {
      ke = (ke + (vel[i + (k * nd)] * vel[i + (k * nd)]));
    }
  }
}
```

OpenMP reduction clause with sum operation

Figure 3: Code snippet of the OpenMP version of the MD code generated using IPT

5. Generating the MPI version of the MD Code using IPT

The steps for generating the MPI version of the MD code using IPT are shown in Figure 4. The questions posed by IPT during the process of generating the MPI version of the MD application are shown in boldface in Figure 4. The details related to the meaning of the questions are available in the IPT documentation.

```
$IPT md.c

NOTE: We currently support only C and C++ programs.

Please select a parallel programming model from the following available options:
1. MPI
2. OpenMP
3. CUDA
1

Please note that by default, the MPI environment initialization functions will be set in
the main function.

Please choose the function that you want to parallelize from the list below:
1 : main
2 : compute
3 : cpu_time
4 : dist
5 : initialize
6 : r8mat_uniform_ab
7 : timestamp
8 : update
2

Please refer to the user-guide for the explanation on each of the patterns, and note that
not all the listed patterns may be relevant for your application type. Please select a
pattern from the following list that best characterizes your parallelization needs:
1. For-Loop Parallelization
2. Stencil
3. Pipeline
4. Data Distribution and Data Collection
5. Data Distribution
6. Data Collection
1

Note: With your response, you will be selecting or declining the parallelization of the
outermost for-loop in the code region shown below. If instead of the outermost for-loop,
there are any inner for-loops in this code region that you are interested in
parallelizing, then, you will be able to select those at a later stage.

for (k = 0; k < np; k++) {
  //Compute the potential energy and forces.
  for (i = 0; i < nd; i++) {
    f[i + (k * nd)] = 0.0;
  }
  for (j = 0; j < np; j++) {
    if (k != j) {
      d = dist(nd,(pos + (k * nd)),(pos + (j * nd)),rij);
      //Attribute half of the potential energy to particle J.
      if (d < PI2) {
        d2 = d;
      }
      else {
        d2 = PI2;
      }
      pe = (pe + (0.5 * pow(sin(d2),2)));
      for (i = 0; i < nd; i++) {
        f[i + (k * nd)] = (f[i + (k * nd)] - ((rij[i] * sin((2.0 * d2))) / d));
      }
    }
  }
  //Compute the kinetic energy.
```

MPI is selected

The `compute` function is selected

The loop shown in Figure 1 is found by IPT

```
  for (i = 0; i < nd; i++) {
    ke = (ke + (vel[i + (k * nd)] * vel[i + (k * nd)]));
  }
}
```

Is this the for loop you are looking for?(y/n)
y

Please specify the type of the data storage (variable, array, structure) from which the data collection should be done at the hotspot for parallelization:
1. Variable/s
2. Array/s
3. Structure/s (not supported currently)
4. Both Variable/s and Array/s
5. Both Variable/s and Structure/s (not supported currently)
6. Variable/s, Array/s, Structure/s (not supported currently)
7. None of the above

4

Please select the arrays from which you want to collect data (format 1,2,3,4 etc. with 1 is the first array in the list, 2 is the second array in the list etc.)
Possible arrays are:
1. f type is double []
2. pos type is double []
3. rij type is double [3UL]
4. vel type is double []
5. pot type is double *
6. kin type is double *

1

These specifications will result in the gather operation

For array [f]
Would you like to collect data from:
1. Distributed stencil array
2. Partially calculated value of an array in a for loop

2

Is this a
1. 1-D array
2. 2-D array

1

Please enter the size of the array: nd*np

Please select the variables to perform the reduce operation on (format: 1,2,3,4 etc.).
List of possible variables are:
1. nd type is int
2. k type is int
3. np type is int
4. d type is double
5. PI2 type is double
6. d2 type is double
7. pe type is double
8. ke type is double

7,8

`ke` and `pe` are selected for reduction

Please select the reduce operation to use for variable [pe]
1. Sum
2. Product
3. Min
4. Max
5. Logical and
6. Bit-wise and
7. Logical or
8. Bit-wise or
9. Logical xor
10. Bit-wise xor
11. Max value and location
12. Min value and location

1

Reduction with sum operation

Would you like to send the results after reducing the chosen variable to all processes or to only one?(1. all 0. one).

```
Note: if option "0" is chosen then only one MPI process will have the combined results.
Please refer to the user-guide for further information on this.
1

Please select the reduce operation to use for variable [ke]
1.  Sum
2.  Product
3.  Min
4.  Max
5.  Logical and
6.  Bit-wise and
7.  Logical or
8.  Bit-wise or
9.  Logical xor
10.  Bit-wise xor
11.  Max value and location
12.  Min value and location
1

Would you like to send the results after reducing the chosen variable to all processes or
to only one?(1. all 0. one).
Note: if option "0" is chosen then only one MPI process will have the combined results.
Please refer to the user-guide for further information on this.
1

Would you like to do this MPI pattern again?(Y/N)
n

Are you writing anything?(Y/N)
n

Running Consistency Tests
```

Results will be shared with all the MPI processes

Figure 4: Steps for generating the MPI version of the MD code using IPT

After the user follows the steps shown in Figure 4, IPT generates an output file named **`rose_md_MPI.c`** in the current working directory. A snippet of the MPI code generated by IPT is shown in Figure 5 and the code inserted by IPT is highlighted in boldface. The complete code is provided in the appendix. As can be noticed from the MPI version of the code in the appendix, IPT prepares and inserts the calls for the `MPI_Allreduce` and `MPI_Allgatherv` functions. It inserts additional statements in the code for calculating the number of elements to be gathered from each MPI process and the displacement of the elements in the result array. It modifies the start and stop conditions of the for-loop that is selected for parallelization. IPT also inserts the code for initializing and terminating the MPI execution environment. It extends the variable declaration section as needed and inserts the statement for including the MPI header file.

```
MPI_Comm_rank(MPI_COMM_WORLD,&rose_rank);
MPI_Comm_size(MPI_COMM_WORLD,&rose_size);

pe = 0.0;
ke = 0.0;

int rose_range0 = (np - 0 )/ rose_size;

if (rose_range0 <= 1 ) {
  rose_upper_limit0 = ((rose_rank+1)*1 <= np ) ? (rose_rank+1)*1 : np;
  rose_lower_limit0= rose_rank*1;
}
else {
  if ( rose_rank  > 0 ) {
     rose_lower_limit0 = (rose_rank-1)*rose_range0 + rose_range0 - (((rose_rank-
                                                    1)*rose_range0 + rose_range0 )% 1) + 0 ;
  }else {
     rose_lower_limit0 = 0;
```

Code for calculating the start and stop conditions for each MPI process working on the for-loop

```
  }
  if (rose_rank < rose_size -1)  {
   rose_upper_limit0 =  (rose_rank)*rose_range0 + rose_range0 - (((rose_rank)*rose_range0
                                                          +  rose_range0) % 1) + 0 - 0;
  }else {
    rose_upper_limit0 = np ;
  }
}

int rose_f_range = nd*np /rose_size;
double * rose_temp_f;
int f_displacement [rose_size];
int f_recvcounts [rose_size];

for ( int rose_index = 0; rose_index < rose_size; rose_index++) {
     f_displacement[rose_index] =  ( rose_index == 0) ? 0 : (rose_index-1)*rose_f_range +
                  rose_f_range - (((rose_index-1)*rose_f_range + rose_f_range )% 1) + 0 ;
}
for ( int rose_index = 0; rose_index < rose_size; rose_index++) {
  if (rose_index < rose_size -1 )  {
   f_recvcounts [rose_index] = f_displacement[rose_index+1] - f_displacement[rose_index];
  }else {
   f_recvcounts [rose_index] = nd*np - f_displacement[rose_index];
  }
}

rose_temp_f = (double *) malloc (sizeof(double)*f_recvcounts[rose_rank]);

for (k = rose_lower_limit0; k < rose_upper_limit0; k++) {
    //Compute the potential energy and forces.
    for (i = 0; i < nd; i++) {
      f[i + (k * nd)] = 0.0;
    }
    for (j = 0; j < np; j++) {
      if (k != j) {
        d = dist(nd,(pos + (k * nd)),(pos + (j
* nd)),rij);
        //Attribute half of the potential energy to particle J.
        if (d < PI2) {
          d2 = d;
        }
        else {
          d2 = PI2;
        }
        pe = (pe + (0.5 * pow(sin(d2),2)));
        for (i = 0; i < nd; i++) {
          f[i + (k * nd)] = (f[i + (k * nd)] - ((rij[i] *
sin((2.0 * d2))) / d));
        }
      }
    }
    //Compute the kinetic energy.
    for (i = 0; i < nd; i++) {
      ke = (ke + (vel[i + (k * nd)] * vel[i + (k * nd)]));
    }
}

MPI_Allreduce(&ke,&rose_ke0,1,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
ke = rose_ke0;

MPI_Allreduce(&pe,&rose_pe0,1,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
pe = rose_pe0;

for (int rose_index =0; rose_index < (f_recvcounts[rose_rank] ); rose_index++) {
     rose_temp_f[rose_index] = f[rose_index + f_displacement[rose_rank]];
}
MPI_Allgatherv(rose_temp_f,f_recvcounts[rose_rank],MPI_DOUBLE,f,f_recvcounts,
                                                 f_displacement,MPI_DOUBLE,MPI_COMM_WORLD);
```

Code for calculating the number of elements to be gathered from each MPI process and the displacement of the elements in the global/result array

Start and stop conditions of the for-loop changed

Calls to MPI_Allreduce and MPI_Allgatherv inserted

Figure 5: Code snippet of the MPI version of the MD code generated using IPT

6. Generating the CUDA version of the MD Code using IPT

The steps for generating the CUDA version of the MD code using IPT are shown in Figure 6, and the questions posed by IPT during the process of generating the CUDA version of the MD application are shown in boldface. The details related to the meaning of the questions are available in the IPT documentation.

```
$IPT md.c

NOTE: We currently support only C and C++ programs.

Please select a parallel programming model from the following available options:
1. MPI
2. OpenMP
3. CUDA
3

Please choose the function in which you wish to insert the kernel call (or parallelize
the for-loop).
1 : main
2 : compute
3 : cpu_time
4 : dist
5 : initialize
6 : r8mat_uniform_ab
7 : timestamp
8 : update
2

Would you like to parallelize
1. For-loop
2. TBD
1

Note: With your response, you will be selecting or declining the parallelization of the
outermost for-loop in the code region shown below. If instead of the outermost for-loop,
there are any inner for-loops in this code region that you are interested in
parallelizing, then, you will be able to select those at a later stage.

for (k = 0; k < np; k++) {
  //Compute the potential energy and forces.
  for (i = 0; i < nd; i++) {
    f[i + (k * nd)] = 0.0;
  }
  for (j = 0; j < np; j++) {
    if (k != j) {
      d = dist(nd,(pos + (k * nd)),(pos + (j * nd)),rij);
      //Attribute half of the potential energy to particle J.
      if (d < PI2) {
        d2 = d;
      }
      else {
        d2 = PI2;
      }
      pe = (pe + (0.5 * pow(sin(d2),2)));
      for (i = 0; i < nd; i++) {
        f[i + (k * nd)] = (f[i + (k * nd)] - ((rij[i] * sin((2.0 * d2))) / d));
      }
    }
  }
  //Compute the kinetic energy.
  for (i = 0; i < nd; i++) {
    ke = (ke + (vel[i + (k * nd)] * vel[i + (k * nd)]));
  }
}
Is this the for loop you are looking for?(y/n)
y

Do you want to perform reduction on any variables? (Y/N)
```

The loop shown in Figure 1 is found by IPT

```
y

Please enter the variables to reduce ([format:  1,2,3 etc. ] where 1 is for the first
variable, 2 is for the second variable and so on). Possible variables to reduce are:
1. nd type is int
2. k type is int
3. np type is int
4. d type is double
5. PI2 type is double
6. d2 type is double
7. pe type is double
8. ke type is double
7,8

Please enter the reduction operation for variable [pe]. Possible reduction operations
are:
1. Sum
2. Product
1

Please enter the reduction operation for variable [ke]. Possible reduction operations
are:
1. Sum
2. Product
1

Is the following array [ f ]
1. Input , 2. Output 3. Input/Output 4. Neither Input nor Output
3
Please specify the dimensions for this variable (format: [ D1,D2,D3, etc. ] with D1 is
the first dimensions, D2 is the second dimensions and so on). Enter 1 if this is not an
array.
np*nd

Is the following array [ pos ]
1. Input , 2. Output 3. Input/Output 4. Neither Input nor Output
1
Please specify the dimensions for this variable (format: [ D1,D2,D3, etc. ] with D1 is
the first dimensions, D2 is the second dimensions and so on). Enter 1 if this is not an
array.
np*nd

Is the following array [ rij ]
1. Input , 2. Output 3. Input/Output 4. Neither Input nor Output
4

Is the following array [ vel ]
1. Input , 2. Output 3. Input/Output 4. Neither Input nor Output
1
Please specify the dimensions for this variable (format: [ D1,D2,D3, etc. ] with D1 is
the first dimensions, D2 is the second dimensions and so on). Enter 1 if this is not an
array.
nd*np

Running Consistency Tests
```

`ke` and `pe` are selected for reduction

Reduction with sum operation

Checking if the data needs to be copied to the device (GPU) and back to the host CPU

Figure 6: Steps for generating the CUDA version of the MD code using IPT

After the user follows the steps shown in Figure 6, IPT generates an output file named **`rose_md.cu`** in the current working directory. A snippet of the CUDA code generated by IPT is shown in Figure 7 and the code inserted by IPT is highlighted in boldface. The complete code is provided in the appendix. As can be noticed from the CUDA code in the appendix, IPT reengineers the serial code to create and insert the kernel and device functions, do memory management, perform the reduction operation, and calculate the number of threads using which the CUDA program can be run. It replaces the for-loop selected for parallelization with the call to the CUDA kernel. It also inserts comments in

the code to inform the user about the purpose of the changes made to their original code.

```
//Inserting code for memory allocation grid size and block size calculation
host_pe= (double*)malloc((1)*((int) ((np - 0 ) / (1)))*sizeof(double));
cudaMalloc((void **) &device_pe,( 1)*((int) ((np - 0 ) / (1)))*sizeof(double));

//Inserting code for memory allocation grid size and block size calculation
host_ke= (double*)malloc((1)*((int) ((np - 0 ) / (1)))*sizeof(double));
cudaMalloc((void **) &device_ke,( 1)*((int) ((np - 0 ) / (1)))*sizeof(double));
```

IPT inserts code for memory management on the GPU & calculates the number of threads in each block of a grid using which the code will run on the GPU

```
//Please note this is the section wherein the number of blocks and threads are
calculated.  To change the number of threads alter the dimBlock whereas to change the
number of blocks alter the dimGrid
int D_rows = ((int) ((np - 0 ) / (1)) > 1024 ) ? (int) ((np - 0 ) / (1))/1024 : (int)
((np - 0 ) / (1));
int D_cols = ((int) ((np - 0 ) / (1)) > 1024 ) ? 1024 : 1;
dim3 dimGrid(D_rows,1);
dim3 dimBlock(D_cols,1);

cudaMalloc((void **) &device_f,(np*nd)*sizeof(double));
cudaMemcpy(device_f,f,(np*nd)*sizeof(double),cudaMemcpyHostToDevice);
cudaMalloc((void **) &device_pos,(np*nd)*sizeof(double));
cudaMemcpy(device_pos,pos,(np*nd)*sizeof(double),cudaMemcpyHostToDevice);
cudaMalloc((void **) &device_vel,(nd*np)*sizeof(double));
cudaMemcpy(device_vel,vel,(nd*np)*sizeof(double),cudaMemcpyHostToDevice);

kernel0<<<dimGrid,dimBlock>>>(device_pe,device_ke,device_f,device_pos,device_vel,np,i,nd,
j,d,PI2,d2,1,(int) ((np - 0 ) / (1)));
```

The loop shown in Figure 1 is replaced by the call to a kernel named **`kernel0`**

```
/*
  int IPT_function_replace;
*/

//Copying from Device to Host
cudaMemcpy(host_pe,device_pe,((int) ((np - 0 ) / (1)))*sizeof(double),
cudaMemcpyDeviceToHost);
```

IPT inserts code for reduction

```
//code for variable reduction
 for(long row = 0; row< 1; ++row){
    for(long col = 0; col <(int)((np - 0 ) / (1)); ++col){
      total_pe+= host_pe[row*(int) ((np - 0 ) / (1))+ col];
    }
 }
 pe+= total_pe;

//Copying from Device to Host
cudaMemcpy(host_ke,device_ke,((int) ((np - 0 ) / (1)))*sizeof(double),
cudaMemcpyDeviceToHost);
 //code for variable reduction
 for(long row = 0; row < 1; ++row){
    for(long col = 0; col < (int) ((np - 0 ) / (1)); ++col){
       total_ke+= host_ke[row*(int) ((np - 0 ) / (1))+ col];
  }
 }
 ke+= total_ke;
 cudaMemcpy(f,device_f,(np*nd)*sizeof(double), cudaMemcpyDeviceToHost);
 cudaFree(device_f);
 cudaFree(device_pos);
 cudaFree(device_vel);
```

Figure 7: Code snippet of the CUDA version of the MD code generated using IPT

7. Compiling & Running the Different Versions of the MD Application

Compiling and running the serial verison
The serial version of the MD application can be compiled using the command:

```
$ icc -o md md.c
```

The executable "md" of the serial version can be run as follows on a compute node:
```
./md 2 1000 1000 0.01
```

In the aforementioned command, we provided 4 arguments that specify:
```
ND, the spatial dimension, which is 2
NP, the number of particles in the simulation, which is 1000
STEP_NUM, the number of time steps, which is 1000
DT, the size of each time step, is which 0.010000
```

Compiling and running the OpenMP verison
Following is the command to compile the OpenMP version of the MD code that was generated by IPT (rose_md_OpenMP.c):

```
$ icc -qopenmp -o rose_md_OpenMP rose_md_OpenMP.c
```

Before running the executable rose_md_OpenMP, we will need to set an environment variable named OMP_NUM_THREADS to specify the number of threads that the executable should run with. With the command shown below, we set the number of threads to 8.

```
$ export OMP_NUM_THREADS=8
```

The executable of the OpenMP version of the code can be run on a compute node in the same way as the serial version:
```
./rose_md_OpenMP 2 1000 1000 0.01
```

It should be noted that the MD code prints its run-time on the standard output. It is programmed to report the "cpu time". **In the case of an OpenMP program, when the "cpu time" is reported, it is the sum total of the time spent by each thread in running the code or the cpu utilization time. Therefore, the execution time reported by the code can look very close to the time taken to run the serial version of the code.** The actual run-time of the OpenMP program can be found by using the time command and by checking the "real" time reported by the command:

```
time ./rose_md_OpenMP 2 1000 1000 0.01
```

Compiling and running the MPI verison
The MPI version of the MD code can be compiled using the mpicxx command as shown below:
```
mpicxx -o rose_md_mpi rose_md_MPI.c
```

The executable can be run on a compute node as follows using the ibrun command that is available on the Stampede supercomputer at TACC:

```
ibrun -np 8 rose_md_mpi 2 1000 1000 0.01
```

If you are not using Stampede, then please refer to the user manual of your system or the MPI library documentation to find the command to use instead of `ibrun` (*e.g.* of other commands: `mpirun`, and `mpiexec`).

While running the MPI version of the program, each MPI process may be printing the content to the standard output such that they end up stepping on each other, thus, resulting in jumbled output. To avoid this from happening, you can select the option of writing any output with only one MPI process at the time of code generation. You can do this by selecting "y" in response to the following question presented by IPT: "`Are you writing anything?(Y/N).`"

Compiling and running the CUDA verison

To compile and run the CUDA code generated by IPT, you would need access to a GPU. If you are using the Stampede 1.0 system, then you may access the GPU through the "gpu" or "gpudev" queues. You would also need to load the "cuda" module when working on Stampede 1.0.

The following commands show the steps for requesting access to a compute node in the "gpu" queue on Stampede 1.0, loading the "cuda" module, and compiling the code using the `nvcc` command:

```
$ idev -p gpudev
$ ml cuda
$ nvcc -o rose_md_cuda rose_md.cu
```

or

```
$ nvcc -arch=compute_35 -code=sm_35 -o rose_md_cuda rose_md.cu
```

The executable can be run as follows on the compute node in the "gpu" queue:

```
./rose_md_cuda 2 1000 1000 0.01
```

8. Performance Comparison of the Different Versions of the MD Code

The run-time comparison of the serial and the three parallel versions of the MD code are shown in Table 1. The code was run using the same arguments (spatial dimensions, number of particles, size of the time-step, and the number of time-steps) across all the versions. The time shown is in seconds.

Table 1. Run-time comparison (the time shown is in seconds)

Arguments Used	Serial	OpenMP (Using 8 threads)	MPI (Using 8 processes)	CUDA (# of threads = # of particles)
2, 500, 500, 0.01	6.831	0.957	3.031	4.874
2, 1000, 1000, 0.01	54.037	6.732	8.446	23.180

We ran the MD application with very small problem sizes. As can be noticed from the data in Table 1, the OpenMP version performed best. This version could take advantage of multiple cores on a node and the shared memory available on it. As compared to the OpenMP version, the MPI version involves larger overheads, mainly for MPI environment initialization and data exchange between MPI processes having separate address space (for the `MPI_Allgatherv` call). The CUDA version involves additional overheads too in copying data to and from the device (GPU) to the host (CPU). In the situation when the computation time is greater than the time spent in the communication and I/O, the MPI version of the code is likely to give best performance.

With the increase in the problem size, the memory required for running the application will also increase. When such a situation arises, distributing the computation across multiple compute nodes and using MPI for data movement across nodes would be the best (or the only possible) option. Using MPI for inter-node communication and OpenMP for intranode communication can also be a viable option.

In essence, it is important to understand the advantages and disadvantages of the different parallel programming models so that one can select the model that may work best for a given application and hardware platform.

9. References:

[1] Molecular Dynamics (MD) Simulation Application. Website accessed on August 10, 2017. https://en.wikipedia.org/wiki/Molecular_dynamics

[2] Molecular Dynamics (MD) code. Website accessed on August 10, 2017. https://people.sc.fsu.edu/~jburkardt/c_src/md/md.html

10. Appendix

1) OpenMP version of the MD code generated using IPT

```
$ cat rose_md_OpenMP.c


# include <omp.h>
# include <stdlib.h>
# include <stdio.h>
# include <time.h>
# include <math.h>
int main(int argc,char *argv[]);
void compute(int np,int nd,double pos[],double vel[],double mass,double f[],double
*pot,double *kin);
double cpu_time();
double dist(int nd,double r1[],double r2[],double dr[]);
void initialize(int np,int nd,double pos[],double vel[],double acc[]);
void r8mat_uniform_ab(int m,int n,double a,double b,int *seed,double r[]);
void timestamp();
void update(int np,int nd,double pos[],double vel[],double f[],double acc[],double
mass,double dt);
/******************************************************************************/

int main(int argc,char *argv[])
/******************************************************************************/
/*
Purpose:
MAIN is the main program for MD.
Discussion:
MD implements a simple molecular dynamics simulation.
```

```
The velocity Verlet time integration scheme is used.
The particles interact with a central pair potential.
This program is based on a FORTRAN90 program by Bill Magro.
Usage:
md nd np step_num dt
where:
nd is the spatial dimension (2 or 3);
np is the number of particles (500, for instance);
step_num is the number of time steps (500, for instance).
dt is the time step (0.1 for instance)
Licensing:
This code is distributed under the GNU LGPL license.
Modified:
27 December 2014
Author:
John Burkardt.
*/
{
double *acc;
double ctime;
double dt;
double e0;
double *force;
int i;
int id;
double kinetic;
double mass = 1.0;
int nd;
int np;
double *pos;
double potential;
int step;
int step_num;
int step_print;
int step_print_index;
int step_print_num;
double *vel;
timestamp();
printf("\n");
printf("MD\n");
printf("  C version\n");
printf("  A molecular dynamics program.\n");
/*
Get the spatial dimension.
*/
if (1 < argc) {
nd = atoi(argv[1]);
}
else {
printf("\n");
printf("  Enter ND, the spatial dimension (2 or 3).\n");
scanf("%d",&nd);
}
//
//  Get the number of particles.
//
if (2 < argc) {
np = atoi(argv[2]);
}
else {
printf("\n");
printf("  Enter NP, the number of particles (500, for instance).\n");
scanf("%d",&np);
}
//
//  Get the number of time steps.
//
if (3 < argc) {
step_num = atoi(argv[3]);
}
else {
```

```
printf("\n");
printf("  Enter ND, the number of time steps (500 or 1000, for instance).\n");
scanf("%d",&step_num);
}
//
//  Get the time steps.
//
if (4 < argc) {
dt = atof(argv[4]);
}
else {
printf("\n");
printf("  Enter DT, the size of the time step (0.1, for instance).\n");
scanf("%g",&dt);
}
/*
Report.
*/
printf("\n");
printf("  ND, the spatial dimension, is %d\n",nd);
printf("  NP, the number of particles in the simulation, is %d\n",np);
printf("  STEP_NUM, the number of time steps, is %d\n",step_num);
printf("  DT, the size of each time step, is %f\n",dt);
/*
Allocate memory.
*/
acc = ((double *)(malloc(((nd * np) * sizeof(double )))));
force = ((double *)(malloc(((nd * np) * sizeof(double )))));
pos = ((double *)(malloc(((nd * np) * sizeof(double )))));
vel = ((double *)(malloc(((nd * np) * sizeof(double )))));
/*
This is the main time stepping loop:
Compute forces and energies,
Update positions, velocities, accelerations.
*/
printf("\n");
printf("  At each step, we report the potential and kinetic energies.\n");
printf("  The sum of these energies should be a constant.\n");
printf("  As an accuracy check, we also print the relative error\n");
printf("  in the total energy.\n");
printf("\n");
printf("      Step      Potential       Kinetic        (P+K-E0)/E0\n");
printf("                Energy P        Energy K       Relative Energy Error\n");
printf("\n");
step_print = 0;
step_print_index = 0;
step_print_num = 10;
ctime = cpu_time();
for (step = 0; step <= step_num; step++) {
if (step == 0) {
initialize(np,nd,pos,vel,acc);
}
else {
update(np,nd,pos,vel,force,acc,mass,dt);
}
compute(np,nd,pos,vel,mass,force,&potential,&kinetic);
if (step == 0) {
e0 = (potential + kinetic);
}
if (step == step_print) {
printf("  %8d  %14f  %14f  %14e\n",step,potential,kinetic,(((potential + kinetic) - e0) /
e0));
step_print_index = (step_print_index + 1);
step_print = ((step_print_index * step_num) / step_print_num);
// printf("\nforce = ");
// for (int ttt = 0; ttt < nd*np; ttt++) {
//    printf("%lf ",force[ttt]);
// }
// printf("\n\n");
}
}
```

```
/*
Report timing.
*/
ctime = (cpu_time() - ctime);
printf("\n");
printf("  Elapsed cpu time: %f seconds.\n",ctime);
/*
Free memory.
*/
free(acc);
free(force);
free(pos);
free(vel);
/*
Terminate.
*/
printf("\n");
printf("MD\n");
printf("  Normal end of execution.\n");
printf("\n");
timestamp();
return 0;
}
/******************************************************************************/

void compute(int np,int nd,double pos[],double vel[],double mass,double f[],double
*pot,double *kin)
/******************************************************************************/
/*
Purpose:
COMPUTE computes the forces and energies.
Discussion:
The computation of forces and energies is fully parallel.
The potential function V(X) is a harmonic well which smoothly
saturates to a maximum value at PI/2:
v(x) = ( sin ( min ( x, PI/2 ) ) )^2
The derivative of the potential is:
dv(x) = 2.0 * sin ( min ( x, PI/2 ) ) * cos ( min ( x, PI/2 ) )
= sin ( 2.0 * min ( x, PI/2 ) )
Licensing:
This code is distributed under the GNU LGPL license.
Modified:
21 November 2007
Author:
John Burkardt.
Parameters:
Input, int NP, the number of particles.
Input, int ND, the number of spatial dimensions.
Input, double POS[ND*NP], the positions.
Input, double VEL[ND*NP], the velocities.
Input, double MASS, the mass of each particle.
Output, double F[ND*NP], the forces.
Output, double *POT, the total potential energy.
Output, double *KIN, the total kinetic energy.
*/
{
double d;
double d2;
int i;
int j;
int k;
double ke;
double pe;
double PI2 = 3.141592653589793 / 2.0;
double rij[3UL];
pe = 0.0;
ke = 0.0;

#pragma omp parallel default(none) shared(pe,ke,np,f,pos,vel,nd,PI2) private(k,i,j,d,d2)
firstprivate(rij)
{
```

```
#pragma omp for reduction ( + :pe,ke)
for (k = 0; k < np; k++) {
 //Compute the potential energy and forces.
  for (i = 0; i < nd; i++) {
    f[i + (k * nd)] = 0.0;
  }
  for (j = 0; j < np; j++) {
    if (k != j) {
      d = dist(nd,(pos + (k * nd)),(pos + (j * nd)),rij);
      //Attribute half of the potential energy to particle J.
      if (d < PI2) {
       d2 = d;
      }
      else {
       d2 = PI2;
      }
      pe = (pe + (0.5 * pow(sin(d2),2)));
      for (i = 0; i < nd; i++) {
        f[i + (k * nd)] = (f[i + (k * nd)] - ((rij[i] * sin((2.0 * d2))) / d));
      }
   }
 }
 //Compute the kinetic energy.
 for (i = 0; i < nd; i++) {
     ke = (ke + (vel[i + (k * nd)] * vel[i + (k * nd)]));
 }
}
}
ke = ((ke * 0.5) * mass);
*pot = pe;
*kin = ke;
}
/**************************************************************************************/

double cpu_time()
/**************************************************************************************/
/*
Purpose:

CPU_TIME reports the total CPU time for a program.
Licensing:
This code is distributed under the GNU LGPL license.
Modified:
27 September 2005
Author:
John Burkardt
Parameters:
Output, double CPU_TIME, the current total elapsed CPU time in second.
*/
{
double value;
value = (((double )(clock())) / ((double )10000001));
return value;
}
/**************************************************************************************/

double dist(int nd,double r1[],double r2[],double dr[])
/**************************************************************************************/
/*
Purpose:
DIST computes the displacement (and its norm) between two particles.
Licensing:
This code is distributed under the GNU LGPL license.
Modified:
21 November 2007
Author:
John Burkardt.
Parameters:
Input, int ND, the number of spatial dimensions.
Input, double R1[ND], R2[ND], the positions of the particles.
```

```
Output, double DR[ND], the displacement vector.
Output, double D, the Euclidean norm of the displacement.
*/
{
double d;
int i;
d = 0.0;
for (i = 0; i < nd; i++) {
dr[i] = (r1[i] - r2[i]);
d = (d + (dr[i] * dr[i]));
}
d = sqrt(d);
return d;
}
/******************************************************************************/

void initialize(int np,int nd,double pos[],double vel[],double acc[])
/******************************************************************************/
/*
Purpose:
INITIALIZE initializes the positions, velocities, and accelerations.
Licensing:
This code is distributed under the GNU LGPL license.
Modified:
26 December 2014
Author:
John Burkardt.
Parameters:
Input, int NP, the number of particles.
Input, int ND, the number of spatial dimensions.
Output, double POS[ND*NP], the positions.
Output, double VEL[ND*NP], the velocities.
Output, double ACC[ND*NP], the accelerations.
*/
{
int i;
int j;
int seed;
/*
Set positions.
*/
seed = 123456789;
r8mat_uniform_ab(nd,np,0.0,10.0,&seed,pos);
/*
Set velocities.
*/
for (j = 0; j < np; j++) {
for (i = 0; i < nd; i++) {
vel[i + (j * nd)] = 0.0;
}
}
/*
Set accelerations.
*/
for (j = 0; j < np; j++) {
for (i = 0; i < nd; i++) {
acc[i + (j * nd)] = 0.0;
}
}
}
/******************************************************************************/

void r8mat_uniform_ab(int m,int n,double a,double b,int *seed,double r[])
/******************************************************************************/
/*
Purpose:
R8MAT_UNIFORM_AB returns a scaled pseudorandom R8MAT.
Discussion:
This routine implements the recursion
seed = 16807 * seed mod ( 2^31 - 1 )
unif = seed / ( 2^31 - 1 )
```

```
The integer arithmetic never requires more than 32 bits,
including a sign bit.
Licensing:
This code is distributed under the GNU LGPL license.
Modified:
03 October 2005
Author:
John Burkardt
Reference:
Paul Bratley, Bennett Fox, Linus Schrage,
A Guide to Simulation,
Second Edition,
Springer, 1987,
ISBN: 0387964673,
LC: QA76.9.C65.B73.
Bennett Fox,
Algorithm 647:
Implementation and Relative Efficiency of Quasirandom
Sequence Generators,
ACM Transactions on Mathematical Software,
Volume 12, Number 4, December 1986, pages 362-376.
Pierre L'Ecuyer,
Random Number Generation,
in Handbook of Simulation,
edited by Jerry Banks,
Wiley, 1998,
ISBN: 0471134031,
LC: T57.62.H37.
Peter Lewis, Allen Goodman, James Miller,
A Pseudo-Random Number Generator for the System/360,
IBM Systems Journal,
Volume 8, Number 2, 1969, pages 136-143.
Parameters:
Input, int M, N, the number of rows and columns.
Input, double A, B, the limits of the pseudorandom values.
Input/output, int *SEED, the "seed" value.  Normally, this
value should not be 0.  On output, SEED has
been updated.
Output, double R[M*N], a matrix of pseudorandom values.
*/
{
int i;
const int i4_huge = 2147483647;
int j;
int k;
if ( *seed == 0) {
fprintf(stderr,"\n");
fprintf(stderr,"R8MAT_UNIFORM_AB - Fatal error!\n");
fprintf(stderr,"  Input value of SEED = 0.\n");
exit(1);
}
for (j = 0; j < n; j++) {
for (i = 0; i < m; i++) {
k = ( *seed / 127773);
*seed = ((16807 * ( *seed - (k * 127773))) - (k * 2836));
if ( *seed < 0) {
*seed = ( *seed + i4_huge);
}
r[i + (j * m)] = (a + (((b - a) * ((double )( *seed))) * 4.656612875E-10));
}
}
}
/******************************************************************************/

void timestamp()
/******************************************************************************/
/*
Purpose:
TIMESTAMP prints the current YMDHMS date as a time stamp.
Example:
31 May 2001 09:45:54 AM
```

```
Licensing:
This code is distributed under the GNU LGPL license.
Modified:
24 September 2003
Author:
John Burkardt
Parameters:
None
*/
{
# define TIME_SIZE 40
static char time_buffer[40UL];
const struct tm *tm;
size_t len;
time_t now;
now = time(0);
tm = (localtime((&now)));
len = strftime(time_buffer,40,"%d %B %Y %I:%M:%S %p",tm);
printf("%s\n",time_buffer);
# undef TIME_SIZE
}
/******************************************************************************/

void update(int np,int nd,double pos[],double vel[],double f[],double acc[],double
mass,double dt)
/******************************************************************************/
/*
Purpose:
UPDATE updates positions, velocities and accelerations.
Discussion:
The time integration is fully parallel.
A velocity Verlet algorithm is used for the updating.
x(t+dt) = x(t) + v(t) * dt + 0.5 * a(t) * dt * dt
v(t+dt) = v(t) + 0.5 * ( a(t) + a(t+dt) ) * dt
a(t+dt) = f(t) / m
Licensing:
This code is distributed under the GNU LGPL license.
Modified:
21 November 2007
Author:
John Burkardt.
Parameters:
Input, int NP, the number of particles.
Input, int ND, the number of spatial dimensions.
Input/output, double POS[ND*NP], the positions.
Input/output, double VEL[ND*NP], the velocities.
Input, double F[ND*NP], the forces.
Input/output, double ACC[ND*NP], the accelerations.
Input, double MASS, the mass.
Input, double DT, the time step.
*/
{
int i;
int j;
double rmass;
rmass = (1.0 / mass);
for (j = 0; j < np; j++) {
for (i = 0; i < nd; i++) {
pos[i + (j * nd)] = ((pos[i + (j * nd)] + (vel[i + (j * nd)] * dt)) + (((0.5 * acc[i + (j
* nd)]) * dt) * dt));
vel[i + (j * nd)] = (vel[i + (j * nd)] + ((0.5 * dt) * ((f[i + (j * nd)] * rmass) + acc[i
+ (j * nd)])));
acc[i + (j * nd)] = (f[i + (j * nd)] * rmass);
}
}
}
```

2) MPI version of the MD code generated using IPT

```
$ cat rose_md_MPI.c


# include <stdlib.h>
# include <stdio.h>
# include <time.h>
# include <math.h>
#include <mpi.h>
int main(int argc,char *argv[]);
void compute(int np,int nd,double pos[],double vel[],double mass,double f[],double
*pot,double *kin);
double cpu_time();
double dist(int nd,double r1[],double r2[],double dr[]);
void initialize(int np,int nd,double pos[],double vel[],double acc[]);
void r8mat_uniform_ab(int m,int n,double a,double b,int *seed,double r[]);
void timestamp();
void update(int np,int nd,double pos[],double vel[],double f[],double acc[],double
mass,double dt);
/******************************************************************************/

int main(int argc,char *argv[])
/******************************************************************************/
/*
  Purpose:
    MAIN is the main program for MD.
  Discussion:
    MD implements a simple molecular dynamics simulation.
    The velocity Verlet time integration scheme is used.
    The particles interact with a central pair potential.
    This program is based on a FORTRAN90 program by Bill Magro.
  Usage:
    md nd np step_num dt
    where:
    * nd is the spatial dimension (2 or 3);
    * np is the number of particles (500, for instance);
    * step_num is the number of time steps (500, for instance).
    * dt is the time step (0.1 for instance)
  Licensing:
    This code is distributed under the GNU LGPL license.
  Modified:
    27 December 2014
  Author:
    John Burkardt.
*/
{
  int rose_size;
  int rose_rank;
  double *acc;
  double ctime;
  double dt;
  double e0;
  double *force;
  int i;
  int id;
  double kinetic;
  double mass = 1.0;
  int nd;
  int np;
  double *pos;
  double potential;
  int step;
  int step_num;
  int step_print;
  int step_print_index;
  int step_print_num;
  double *vel;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rose_rank);
  MPI_Comm_size(MPI_COMM_WORLD,&rose_size);
  timestamp();
  printf("\n");
```

```
  printf("MD\n");
  printf("  C version\n");
  printf("  A molecular dynamics program.\n");
/*
  Get the spatial dimension.
*/
  if (1 < argc) {
    nd = atoi(argv[1]);
  }
  else {
    printf("\n");
    printf("  Enter ND, the spatial dimension (2 or 3).\n");
    scanf("%d",&nd);
  }
//
//  Get the number of particles.
//
  if (2 < argc) {
    np = atoi(argv[2]);
  }
  else {
    printf("\n");
    printf("  Enter NP, the number of particles (500, for instance).\n");
    scanf("%d",&np);
  }
//
//  Get the number of time steps.
//
  if (3 < argc) {
    step_num = atoi(argv[3]);
  }
  else {
    printf("\n");
    printf("  Enter ND, the number of time steps (500 or 1000, for instance).\n");
    scanf("%d",&step_num);
  }
//
//  Get the time steps.
//
  if (4 < argc) {
    dt = atof(argv[4]);
  }
  else {
    printf("\n");
    printf("  Enter DT, the size of the time step (0.1, for instance).\n");
    scanf("%g",&dt);
  }
/*
  Report.
*/
  printf("\n");
  printf("  ND, the spatial dimension, is %d\n",nd);
  printf("  NP, the number of particles in the simulation, is %d\n",np);
  printf("  STEP_NUM, the number of time steps, is %d\n",step_num);
  printf("  DT, the size of each time step, is %f\n",dt);
/*
  Allocate memory.
*/
  acc = ((double *)(malloc(((nd * np) * sizeof(double )))));
  force = ((double *)(malloc(((nd * np) * sizeof(double )))));
  pos = ((double *)(malloc(((nd * np) * sizeof(double )))));
  vel = ((double *)(malloc(((nd * np) * sizeof(double )))));
/*
  This is the main time stepping loop:
    Compute forces and energies,
    Update positions, velocities, accelerations.
*/
  printf("\n");
  printf("  At each step, we report the potential and kinetic energies.\n");
  printf("  The sum of these energies should be a constant.\n");
  printf("  As an accuracy check, we also print the relative error\n");
```

```
  printf("  in the total energy.\n");
  printf("\n");
  printf("      Step      Potential       Kinetic        (P+K-E0)/E0\n");
  printf("                Energy P        Energy K       Relative Energy Error\n");
  printf("\n");
  step_print = 0;
  step_print_index = 0;
  step_print_num = 10;
  ctime = cpu_time();
  for (step = 0; step <= step_num; step++) {
    if (step == 0) {
      initialize(np,nd,pos,vel,acc);
    }
    else {
      update(np,nd,pos,vel,force,acc,mass,dt);
    }
    compute(np,nd,pos,vel,mass,force,&potential,&kinetic);
    if (step == 0) {
      e0 = (potential + kinetic);
    }
    if (step == step_print) {
      printf("  %8d  %14f  %14f  %14e\n",step,potential,kinetic,(((potential + kinetic) -
e0) / e0));
      step_print_index = (step_print_index + 1);
      step_print = ((step_print_index * step_num) / step_print_num);
// printf("\nforce = ");
// for (int ttt = 0; ttt < nd*np; ttt++) {
//    printf("%lf ",force[ttt]);
// }
// printf("\n\n");
    }
  }
/*
  Report timing.
*/
  ctime = (cpu_time() - ctime);
  printf("\n");
  printf("  Elapsed cpu time: %f seconds.\n",ctime);
/*
  Free memory.
*/
  free(acc);
  free(force);
  free(pos);
  free(vel);
/*
  Terminate.
*/
  printf("\n");
  printf("MD\n");
  printf("  Normal end of execution.\n");
  printf("\n");
  timestamp();
  MPI_Finalize();
  return 0;
}
/******************************************************************************/

void compute(int np,int nd,double pos[],double vel[],double mass,double f[],double
*pot,double *kin)
/******************************************************************************/
/*
  Purpose:
    COMPUTE computes the forces and energies.
  Discussion:
    The computation of forces and energies is fully parallel.
    The potential function V(X) is a harmonic well which smoothly
    saturates to a maximum value at PI/2:
      v(x) = ( sin ( min ( x, PI/2 ) ) )^2
    The derivative of the potential is:
      dv(x) = 2.0 * sin ( min ( x, PI/2 ) ) * cos ( min ( x, PI/2 ) )
```

```
            = sin ( 2.0 * min ( x, PI/2 ) )
  Licensing:
    This code is distributed under the GNU LGPL license.
  Modified:
    21 November 2007
  Author:
    John Burkardt.
  Parameters:
    Input, int NP, the number of particles.
    Input, int ND, the number of spatial dimensions.
    Input, double POS[ND*NP], the positions.
    Input, double VEL[ND*NP], the velocities.
    Input, double MASS, the mass of each particle.
    Output, double F[ND*NP], the forces.
    Output, double *POT, the total potential energy.
    Output, double *KIN, the total kinetic energy.
*/
{
  double rose_ke0;
  double rose_pe0;
  int rose_lower_limit0;
  int rose_upper_limit0;
  int rose_size;
  int rose_rank;
  double d;
  double d2;
  int i;
  int j;
  int k;
  double ke;
  double pe;
  double PI2 = 3.141592653589793 / 2.0;
  double rij[3UL];
  MPI_Comm_rank(MPI_COMM_WORLD,&rose_rank);
  MPI_Comm_size(MPI_COMM_WORLD,&rose_size);
  pe = 0.0;
  ke = 0.0;
  int rose_range0 = (np - 0 )/ rose_size;
  if (rose_range0 <= 1 ) {
    rose_upper_limit0 = ((rose_rank+1)*1 <= np ) ? (rose_rank+1)*1 : np;
    rose_lower_limit0= rose_rank*1;
  }else {
   if ( rose_rank  > 0 ) {
     rose_lower_limit0 = (rose_rank-1)*rose_range0 + rose_range0 - (((rose_rank-
         1)*rose_range0 + rose_range0 )% 1) + 0 ;
   }else {
     rose_lower_limit0 = 0;
   }
   if (rose_rank < rose_size -1)  {
    rose_upper_limit0 =  (rose_rank)*rose_range0 + rose_range0 -
              (((rose_rank)*rose_range0 +  rose_range0) % 1) + 0 - 0;
   }else {
     rose_upper_limit0 = np ;
   }
 }
 int rose_f_range = nd*np /rose_size;
 double * rose_temp_f;
 int f_displacement [rose_size];
 int f_recvcounts [rose_size];
 for ( int rose_index = 0; rose_index < rose_size; rose_index++) {
   f_displacement[rose_index] =  ( rose_index == 0) ? 0 : (rose_index-1)*rose_f_range +

     rose_f_range - (((rose_index-1)*rose_f_range + rose_f_range )% 1) + 0 ;
 }
 for ( int rose_index = 0; rose_index < rose_size; rose_index++) {
  if (rose_index < rose_size -1 )  {
   f_recvcounts [rose_index] = f_displacement[rose_index+1] - f_displacement[rose_index];
  }else {
   f_recvcounts [rose_index] = nd*np - f_displacement[rose_index];
  }
 }
```

```
 rose_temp_f = (double *) malloc (sizeof(double)*f_recvcounts[rose_rank]);
 for (k = rose_lower_limit0; k < rose_upper_limit0; k++) {
   //Compute the potential energy and forces.
  for (i = 0; i < nd; i++) {
    f[i + (k * nd)] = 0.0;
  }
  for (j = 0; j < np; j++) {
    if (k != j) {
      d = dist(nd,(pos + (k * nd)),(pos + (j * nd)),rij);
      //Attribute half of the potential energy to particle J.
      if (d < PI2) {
       d2 = d;
      }
      else {
       d2 = PI2;
      }
      pe = (pe + (0.5 * pow(sin(d2),2)));
      for (i = 0; i < nd; i++) {
        f[i + (k * nd)] = (f[i + (k * nd)] - ((rij[i] * sin((2.0 * d2))) / d));
      }
   }
  }
  //Compute the kinetic energy.
   for (i = 0; i < nd; i++) {
     ke = (ke + (vel[i + (k * nd)] * vel[i + (k * nd)]));
   }
  }
  MPI_Allreduce(&ke,&rose_ke0,1,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
  ke = rose_ke0;
  MPI_Allreduce(&pe,&rose_pe0,1,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
  pe = rose_pe0;
  for (int rose_index =0; rose_index < (f_recvcounts[rose_rank] ); rose_index++) {
    rose_temp_f[rose_index] = f[rose_index + f_displacement[rose_rank]];
  }
  MPI_Allgatherv(rose_temp_f,f_recvcounts[rose_rank],MPI_DOUBLE,f,f_recvcounts,
                              f_displacement,MPI_DOUBLE,MPI_COMM_WORLD);
  ke = ((ke * 0.5) * mass);
   *pot = pe;
   *kin = ke;
}
/******************************************************************************/

double cpu_time()
/******************************************************************************/
/*
  Purpose:

    CPU_TIME reports the total CPU time for a program.
  Licensing:
    This code is distributed under the GNU LGPL license.
  Modified:
    27 September 2005
  Author:
    John Burkardt
  Parameters:
    Output, double CPU_TIME, the current total elapsed CPU time in second.
*/
{
  double value;
  value = (((double )(clock())) / ((double )10000001));
  return value;
}
/******************************************************************************/

double dist(int nd,double r1[],double r2[],double dr[])
/******************************************************************************/
/*
  Purpose:
    DIST computes the displacement (and its norm) between two particles.
  Licensing:
    This code is distributed under the GNU LGPL license.
```

```
  Modified:
    21 November 2007
  Author:
    John Burkardt.
  Parameters:
    Input, int ND, the number of spatial dimensions.
    Input, double R1[ND], R2[ND], the positions of the particles.
    Output, double DR[ND], the displacement vector.
    Output, double D, the Euclidean norm of the displacement.
*/
{
  double d;
  int i;
  d = 0.0;
  for (i = 0; i < nd; i++) {
    dr[i] = (r1[i] - r2[i]);
    d = (d + (dr[i] * dr[i]));
  }
  d = sqrt(d);
  return d;
}
/******************************************************************************/

void initialize(int np,int nd,double pos[],double vel[],double acc[])
/******************************************************************************/
/*
  Purpose:
    INITIALIZE initializes the positions, velocities, and accelerations.
  Licensing:
    This code is distributed under the GNU LGPL license.
  Modified:
    26 December 2014
  Author:
    John Burkardt.
  Parameters:
    Input, int NP, the number of particles.
    Input, int ND, the number of spatial dimensions.
    Output, double POS[ND*NP], the positions.
    Output, double VEL[ND*NP], the velocities.
    Output, double ACC[ND*NP], the accelerations.
*/
{
  int i;
  int j;
  int seed;
/*
  Set positions.
*/
  seed = 123456789;
  r8mat_uniform_ab(nd,np,0.0,10.0,&seed,pos);
/*
  Set velocities.
*/
  for (j = 0; j < np; j++) {
    for (i = 0; i < nd; i++) {
      vel[i + (j * nd)] = 0.0;
    }
  }
/*
  Set accelerations.
*/
  for (j = 0; j < np; j++) {
    for (i = 0; i < nd; i++) {
      acc[i + (j * nd)] = 0.0;
    }
  }
}
/******************************************************************************/

void r8mat_uniform_ab(int m,int n,double a,double b,int *seed,double r[])
/******************************************************************************/
```

```
/*
  Purpose:
    R8MAT_UNIFORM_AB returns a scaled pseudorandom R8MAT.
  Discussion:
    This routine implements the recursion
      seed = 16807 * seed mod ( 2^31 - 1 )
      unif = seed / ( 2^31 - 1 )
    The integer arithmetic never requires more than 32 bits,
    including a sign bit.
  Licensing:
    This code is distributed under the GNU LGPL license.
  Modified:
    03 October 2005
  Author:
    John Burkardt
  Reference:
    Paul Bratley, Bennett Fox, Linus Schrage,
    A Guide to Simulation,
    Second Edition,
    Springer, 1987,
    ISBN: 0387964673,
    LC: QA76.9.C65.B73.
    Bennett Fox,
    Algorithm 647:
    Implementation and Relative Efficiency of Quasirandom
    Sequence Generators,
    ACM Transactions on Mathematical Software,
    Volume 12, Number 4, December 1986, pages 362-376.
    Pierre L'Ecuyer,
    Random Number Generation,
    in Handbook of Simulation,
    edited by Jerry Banks,
    Wiley, 1998,
    ISBN: 0471134031,
    LC: T57.62.H37.
    Peter Lewis, Allen Goodman, James Miller,
    A Pseudo-Random Number Generator for the System/360,
    IBM Systems Journal,
    Volume 8, Number 2, 1969, pages 136-143.
  Parameters:
    Input, int M, N, the number of rows and columns.
    Input, double A, B, the limits of the pseudorandom values.
    Input/output, int *SEED, the "seed" value.  Normally, this
    value should not be 0.  On output, SEED has
    been updated.
    Output, double R[M*N], a matrix of pseudorandom values.
*/
{
  int i;
  const int i4_huge = 2147483647;
  int j;
  int k;
  if ( *seed == 0) {
    fprintf(stderr,"\n");
    fprintf(stderr,"R8MAT_UNIFORM_AB - Fatal error!\n");
    fprintf(stderr,"  Input value of SEED = 0.\n");
    exit(1);
  }
  for (j = 0; j < n; j++) {
    for (i = 0; i < m; i++) {
      k = ( *seed / 127773);
       *seed = ((16807 * ( *seed - (k * 127773))) - (k * 2836));
      if ( *seed < 0) {
         *seed = ( *seed + i4_huge);
      }
      r[i + (j * m)] = (a + (((b - a) * ((double )( *seed))) * 4.656612875E-10));
    }
  }
}
/*****************************************************************************/
```

```
void timestamp()
/******************************************************************************/
/*
  Purpose:
    TIMESTAMP prints the current YMDHMS date as a time stamp.
  Example:
    31 May 2001 09:45:54 AM
  Licensing:
    This code is distributed under the GNU LGPL license.
  Modified:
    24 September 2003
  Author:
    John Burkardt
  Parameters:
    None
*/
{
# define TIME_SIZE 40
  static char time_buffer[40UL];
  const struct tm *tm;
  size_t len;
  time_t now;
  now = time(0);
  tm = (localtime((&now)));
  len = strftime(time_buffer,40,"%d %B %Y %I:%M:%S %p",tm);
  printf("%s\n",time_buffer);
# undef TIME_SIZE
}
/******************************************************************************/

void update(int np,int nd,double pos[],double vel[],double f[],double acc[],double
mass,double dt)
/******************************************************************************/
/*
  Purpose:
    UPDATE updates positions, velocities and accelerations.
  Discussion:
    The time integration is fully parallel.
    A velocity Verlet algorithm is used for the updating.
    x(t+dt) = x(t) + v(t) * dt + 0.5 * a(t) * dt * dt
    v(t+dt) = v(t) + 0.5 * ( a(t) + a(t+dt) ) * dt
    a(t+dt) = f(t) / m
  Licensing:
    This code is distributed under the GNU LGPL license.
  Modified:
    21 November 2007
  Author:
    John Burkardt.
  Parameters:
    Input, int NP, the number of particles.
    Input, int ND, the number of spatial dimensions.
    Input/output, double POS[ND*NP], the positions.
    Input/output, double VEL[ND*NP], the velocities.
    Input, double F[ND*NP], the forces.
    Input/output, double ACC[ND*NP], the accelerations.
    Input, double MASS, the mass.
    Input, double DT, the time step.
*/
{
  int i;
  int j;
  double rmass;
  rmass = (1.0 / mass);
  for (j = 0; j < np; j++) {
    for (i = 0; i < nd; i++) {
      pos[i + (j * nd)] = ((pos[i + (j * nd)] + (vel[i + (j * nd)] * dt)) + (((0.5 *
acc[i + (j * nd)]) * dt) * dt));
      vel[i + (j * nd)] = (vel[i + (j * nd)] + ((0.5 * dt) * ((f[i + (j * nd)] * rmass) +
acc[i + (j * nd)])));
      acc[i + (j * nd)] = (f[i + (j * nd)] * rmass);
    }
```

```
   }
}
```

3) CUDA version of the MD code generated using IPT

```
$cat rose_md.cu

# include <stdlib.h>
# include <stdio.h>
# include <time.h>
# include <math.h>

int main(int argc,char *argv[]);
void compute(int np,int nd,double pos[],double vel[],double mass,double f[],double
*pot,double *kin);
double cpu_time();
double dist(int nd,double r1[],double r2[],double dr[]);
void initialize(int np,int nd,double pos[],double vel[],double acc[]);
void r8mat_uniform_ab(int m,int n,double a,double b,int *seed,double r[]);
void timestamp();
void update(int np,int nd,double pos[],double vel[],double f[],double acc[],double
mass,double dt);


/******************************************************************************/
__device__ double dist_rose(int nd,double r1[],double r2[],double dr[]){
double d;
int i;
d = 0.0;
for(i = 0;i < nd;i++) {dr[i] =(r1[i] - r2[i]);d =(d +(dr[i] * dr[i]));}
d = sqrt(d);
return d;
}

int main(int argc,char *argv[])
/******************************************************************************/
/*
  Purpose:
    MAIN is the main program for MD.
  Discussion:
    MD implements a simple molecular dynamics simulation.
    The velocity Verlet time integration scheme is used.
    The particles interact with a central pair potential.
    This program is based on a FORTRAN90 program by Bill Magro.
  Usage:
    md nd np step_num dt
    where:
    * nd is the spatial dimension (2 or 3);
    * np is the number of particles (500, for instance);
    * step_num is the number of time steps (500, for instance).
    * dt is the time step (0.1 for instance)
  Licensing:
    This code is distributed under the GNU LGPL license.
  Modified:
    27 December 2014
  Author:
    John Burkardt.
*/
{
  double *acc;
  double ctime;
  double dt;
  double e0;
  double *force;
```

```
  int i;
  int id;
  double kinetic;
  double mass = 1.0;
  int nd;
  int np;
  double *pos;
  double potential;
  int step;
  int step_num;
  int step_print;
  int step_print_index;
  int step_print_num;
  double *vel;
  timestamp();
  printf("\n");
  printf("MD\n");
  printf("  C version\n");
  printf("  A molecular dynamics program.\n");
/*
  Get the spatial dimension.
*/
  if (1 < argc) {
    nd = atoi(argv[1]);
  }
  else {
    printf("\n");
    printf("  Enter ND, the spatial dimension (2 or 3).\n");
    scanf("%d",&nd);
  }
//
//  Get the number of particles.
//
  if (2 < argc) {
    np = atoi(argv[2]);
  }
  else {
    printf("\n");
    printf("  Enter NP, the number of particles (500, for instance).\n");
    scanf("%d",&np);
  }
//
//  Get the number of time steps.
//
  if (3 < argc) {
    step_num = atoi(argv[3]);
  }
  else {
    printf("\n");
    printf("  Enter ND, the number of time steps (500 or 1000, for instance).\n");
    scanf("%d",&step_num);
  }
//
//  Get the time steps.
//
  if (4 < argc) {
    dt = atof(argv[4]);
  }
  else {
    printf("\n");
    printf("  Enter DT, the size of the time step (0.1, for instance).\n");
    scanf("%g",&dt);
  }
/*
  Report.
*/
  printf("\n");
  printf("  ND, the spatial dimension, is %d\n",nd);
  printf("  NP, the number of particles in the simulation, is %d\n",np);
  printf("  STEP_NUM, the number of time steps, is %d\n",step_num);
  printf("  DT, the size of each time step, is %f\n",dt);
```

```
/*
  Allocate memory.
*/
  acc = ((double *)(malloc(((nd * np) * sizeof(double )))));
  force = ((double *)(malloc(((nd * np) * sizeof(double )))));
  pos = ((double *)(malloc(((nd * np) * sizeof(double )))));
  vel = ((double *)(malloc(((nd * np) * sizeof(double )))));
/*
  This is the main time stepping loop:
    Compute forces and energies,
    Update positions, velocities, accelerations.
*/
  printf("\n");
  printf("  At each step, we report the potential and kinetic energies.\n");
  printf("  The sum of these energies should be a constant.\n");
  printf("  As an accuracy check, we also print the relative error\n");
  printf("  in the total energy.\n");
  printf("\n");
  printf("      Step      Potential       Kinetic        (P+K-E0)/E0\n");
  printf("                Energy P        Energy K       Relative Energy Error\n");
  printf("\n");
  step_print = 0;
  step_print_index = 0;
  step_print_num = 10;
  ctime = cpu_time();
  for (step = 0; step <= step_num; step++) {
    if (step == 0) {
      initialize(np,nd,pos,vel,acc);
    }
    else {
      update(np,nd,pos,vel,force,acc,mass,dt);
    }
    compute(np,nd,pos,vel,mass,force,&potential,&kinetic);
    if (step == 0) {
      e0 = (potential + kinetic);
    }
    if (step == step_print) {
      printf("  %8d  %14f  %14f  %14e\n",step,potential,kinetic,(((potential + kinetic) -
e0) / e0));
      step_print_index = (step_print_index + 1);
      step_print = ((step_print_index * step_num) / step_print_num);
// printf("\nforce = ");
// for (int ttt = 0; ttt < nd*np; ttt++) {
//    printf("%lf ",force[ttt]);
// }
// printf("\n\n");
    }
  }
/*
  Report timing.
*/
  ctime = (cpu_time() - ctime);
  printf("\n");
  printf("  Elapsed cpu time: %f seconds.\n",ctime);
/*
  Free memory.
*/
  free(acc);
  free(force);
  free(pos);
  free(vel);
/*
  Terminate.
*/
  printf("\n");
  printf("MD\n");
  printf("  Normal end of execution.\n");
  printf("\n");
  timestamp();
  return 0;
}
```

```
/*****************************************************************************/
void __global__ kernel0(double pe[],double ke[],double * f,double * pos,double * vel,int
np,int i,int nd,int j,double d,double PI2,double d2, int device_M , int device_N){
 int k =  blockIdx.x * blockDim.x + threadIdx.x;
 int  print_statement_deleted_here=0;
 double rij[3UL];
 pe[k] = 0;
 ke[k] = 0;
 {
 for(i = 0;i < nd;i++) {
   f[i +(k * nd)] = 0.0;
 }
 for(j = 0;j < np;j++) {
   if(k != j) {
     d = dist_rose(nd,(pos +(k * nd)),(pos +(j * nd)),rij);
     if(d < PI2) {
       d2 = d;
     }else {
       d2 = PI2;
    }
    pe[k] =(pe[k] +(0.5 * pow(sin(d2),2)));
    for(i = 0;i < nd;i++) {
      f[i +(k * nd)] =(f[i +(k * nd)] -((rij[i] * sin((2.0 * d2))) / d));
    }}}
   for(i = 0;i < nd;i++) {ke[k] =(ke[k] +(vel[i +(k * nd)] * vel[i +(k * nd)]));}}
   __syncthreads();
}

void compute(int np,int nd,double pos[],double vel[],double mass,double f[],double
*pot,double *kin)
/*****************************************************************************/
/*
  Purpose:
    COMPUTE computes the forces and energies.
  Discussion:
    The computation of forces and energies is fully parallel.
    The potential function V(X) is a harmonic well which smoothly
    saturates to a maximum value at PI/2:
      v(x) = ( sin ( min ( x, PI/2 ) ) )^2
    The derivative of the potential is:
      dv(x) = 2.0 * sin ( min ( x, PI/2 ) ) * cos ( min ( x, PI/2 ) )
            = sin ( 2.0 * min ( x, PI/2 ) )
  Licensing:
    This code is distributed under the GNU LGPL license.
  Modified:
    21 November 2007
  Author:
    John Burkardt.
  Parameters:
    Input, int NP, the number of particles.
    Input, int ND, the number of spatial dimensions.
    Input, double POS[ND*NP], the positions.
    Input, double VEL[ND*NP], the velocities.
    Input, double MASS, the mass of each particle.
    Output, double F[ND*NP], the forces.
    Output, double *POT, the total potential energy.
    Output, double *KIN, the total kinetic energy.
*/
{
  double *device_vel;
  double *device_pos;
  double *device_f;
  double total_ke = 0;
  double *host_ke;
  double *device_ke;
  double total_pe = 0;
  double *host_pe;
  double *device_pe;
  double d;
  double d2;
  int i;
```

```
  int j;
  int k;
  double ke;
  double pe;
  double PI2 = 3.141592653589793 / 2.0;
  double rij[3UL];
  pe = 0.0;
  ke = 0.0;
//Inserting code for memory allocation grid size and block size calculation
host_pe= (double*)malloc((1)*((int) ((np - 0 ) / (1)))*sizeof(double));
cudaMalloc((void **) &device_pe,( 1)*((int) ((np - 0 ) / (1)))*sizeof(double));
//Inserting code for memory allocation grid size and block size calculation
host_ke= (double*)malloc((1)*((int) ((np - 0 ) / (1)))*sizeof(double));
cudaMalloc((void **) &device_ke,( 1)*((int) ((np - 0 ) / (1)))*sizeof(double));
//Please note this is the section wherein the number of blocks and threads are calculated.
To change the number of threads alter the dimBlock whereas to change the number of blocks
alter the dimGrid
int D_rows = ((int) ((np - 0 ) / (1)) > 1024 ) ? (int) ((np - 0 ) / (1))/1024 : (int) ((np
- 0 ) / (1));
int D_cols = ((int) ((np - 0 ) / (1)) > 1024 ) ? 1024 : 1;
dim3 dimGrid(D_rows,1);
dim3 dimBlock(D_cols,1);
cudaMalloc((void **) &device_f,(np*nd)*sizeof(double));
cudaMemcpy(device_f,f,(np*nd)*sizeof(double),cudaMemcpyHostToDevice);
cudaMalloc((void **) &device_pos,(np*nd)*sizeof(double));
cudaMemcpy(device_pos,pos,(np*nd)*sizeof(double),cudaMemcpyHostToDevice);
cudaMalloc((void **) &device_vel,(nd*np)*sizeof(double));
cudaMemcpy(device_vel,vel,(nd*np)*sizeof(double),cudaMemcpyHostToDevice);
kernel0<<<dimGrid,dimBlock>>>(device_pe,device_ke,device_f,device_pos,device_vel,np,i,nd,j
,d,PI2,d2,1,(int) ((np - 0 ) / (1)));
/*
  int IPT_function_replace;
*/
//Copying from Device to Host
cudaMemcpy(host_pe,device_pe,((int) ((np - 0 ) / (1)))*sizeof(double),
cudaMemcpyDeviceToHost);
 //code for variable reduction
 for(long row = 0; row < 1; ++row){for(long col = 0; col < (int) ((np - 0 ) / (1)); ++col)
{  total_pe+= host_pe[row*(int) ((np - 0 ) / (1))+ col];    }  }
pe+= total_pe;
//  Ending Parallelization
//Copying from Device to Host
cudaMemcpy(host_ke,device_ke,((int) ((np - 0 ) / (1)))*sizeof(double),
cudaMemcpyDeviceToHost);
 //code for variable reduction
 for(long row = 0; row < 1; ++row){for(long col = 0; col < (int) ((np - 0 ) / (1)); ++col)
{  total_ke+= host_ke[row*(int) ((np - 0 ) / (1))+ col];    }  }
ke+= total_ke;
//  Ending Parallelization
cudaMemcpy(f,device_f,(np*nd)*sizeof(double), cudaMemcpyDeviceToHost);
cudaFree(device_f);
cudaFree(device_pos);
cudaFree(device_vel);
  ke = ((ke * 0.5) * mass);
   *pot = pe;
   *kin = ke;
}
/******************************************************************************/

double cpu_time()
/******************************************************************************/
/*
  Purpose:

    CPU_TIME reports the total CPU time for a program.
  Licensing:
    This code is distributed under the GNU LGPL license.
  Modified:
    27 September 2005
  Author:
    John Burkardt
```

```
  Parameters:
    Output, double CPU_TIME, the current total elapsed CPU time in second.
*/
{
  double value;
  value = (((double )(clock())) / ((double )10000001));
  return value;
}
/******************************************************************************/

double dist(int nd,double r1[],double r2[],double dr[])
/******************************************************************************/
/*
  Purpose:
    DIST computes the displacement (and its norm) between two particles.
  Licensing:
    This code is distributed under the GNU LGPL license.
  Modified:
    21 November 2007
  Author:
    John Burkardt.
  Parameters:
    Input, int ND, the number of spatial dimensions.
    Input, double R1[ND], R2[ND], the positions of the particles.
    Output, double DR[ND], the displacement vector.
    Output, double D, the Euclidean norm of the displacement.
*/
{
  double d;
  int i;
  d = 0.0;
  for (i = 0; i < nd; i++) {
    dr[i] = (r1[i] - r2[i]);
    d = (d + (dr[i] * dr[i]));
  }
  d = sqrt(d);
  return d;
}
/******************************************************************************/

void initialize(int np,int nd,double pos[],double vel[],double acc[])
/******************************************************************************/
/*
  Purpose:
    INITIALIZE initializes the positions, velocities, and accelerations.
  Licensing:
    This code is distributed under the GNU LGPL license.
  Modified:
    26 December 2014
  Author:
    John Burkardt.
  Parameters:
    Input, int NP, the number of particles.
    Input, int ND, the number of spatial dimensions.
    Output, double POS[ND*NP], the positions.
    Output, double VEL[ND*NP], the velocities.
    Output, double ACC[ND*NP], the accelerations.
*/
{
  int i;
  int j;
  int seed;
/*
  Set positions.
*/
  seed = 123456789;
  r8mat_uniform_ab(nd,np,0.0,10.0,&seed,pos);
/*
  Set velocities.
*/
  for (j = 0; j < np; j++) {
```

```
    for (i = 0; i < nd; i++) {
      vel[i + (j * nd)] = 0.0;
    }
  }
/*
  Set accelerations.
*/
  for (j = 0; j < np; j++) {
    for (i = 0; i < nd; i++) {
      acc[i + (j * nd)] = 0.0;
    }
  }
}
/******************************************************************************/

void r8mat_uniform_ab(int m,int n,double a,double b,int *seed,double r[])
/******************************************************************************/
/*
  Purpose:
    R8MAT_UNIFORM_AB returns a scaled pseudorandom R8MAT.
  Discussion:
    This routine implements the recursion
      seed = 16807 * seed mod ( 2^31 - 1 )
      unif = seed / ( 2^31 - 1 )
    The integer arithmetic never requires more than 32 bits,
    including a sign bit.
  Licensing:
    This code is distributed under the GNU LGPL license.
  Modified:
    03 October 2005
  Author:
    John Burkardt
  Reference:
    Paul Bratley, Bennett Fox, Linus Schrage,
    A Guide to Simulation,
    Second Edition,
    Springer, 1987,
    ISBN: 0387964673,
    LC: QA76.9.C65.B73.
    Bennett Fox,
    Algorithm 647:
    Implementation and Relative Efficiency of Quasirandom
    Sequence Generators,
    ACM Transactions on Mathematical Software,
    Volume 12, Number 4, December 1986, pages 362-376.
    Pierre L'Ecuyer,
    Random Number Generation,
    in Handbook of Simulation,
    edited by Jerry Banks,
    Wiley, 1998,
    ISBN: 0471134031,
    LC: T57.62.H37.
    Peter Lewis, Allen Goodman, James Miller,
    A Pseudo-Random Number Generator for the System/360,
    IBM Systems Journal,
    Volume 8, Number 2, 1969, pages 136-143.
  Parameters:
    Input, int M, N, the number of rows and columns.
    Input, double A, B, the limits of the pseudorandom values.
    Input/output, int *SEED, the "seed" value.  Normally, this
    value should not be 0.  On output, SEED has
    been updated.
    Output, double R[M*N], a matrix of pseudorandom values.
*/
{
  int i;
  const int i4_huge = 2147483647;
  int j;
  int k;
  if ( *seed == 0) {
    fprintf(stderr,"\n");
```

```
    fprintf(stderr,"R8MAT_UNIFORM_AB - Fatal error!\n");
    fprintf(stderr,"  Input value of SEED = 0.\n");
    exit(1);
  }
  for (j = 0; j < n; j++) {
    for (i = 0; i < m; i++) {
      k = ( *seed / 127773);
       *seed = ((16807 * ( *seed - (k * 127773))) - (k * 2836));
      if ( *seed < 0) {
         *seed = ( *seed + i4_huge);
      }
      r[i + (j * m)] = (a + (((b - a) * ((double )( *seed))) * 4.656612875E-10));
    }
  }
}
/******************************************************************************/

void timestamp()
/******************************************************************************/
/*
  Purpose:
    TIMESTAMP prints the current YMDHMS date as a time stamp.
  Example:
    31 May 2001 09:45:54 AM
  Licensing:
    This code is distributed under the GNU LGPL license.
  Modified:
    24 September 2003
  Author:
    John Burkardt
  Parameters:
    None
*/
{
# define TIME_SIZE 40
  static char time_buffer[40UL];
  const struct tm *tm;
  size_t len;
  time_t now;
  now = time(0);
  tm = (localtime((&now)));
  len = strftime(time_buffer,40,"%d %B %Y %I:%M:%S %p",tm);
  printf("%s\n",time_buffer);
# undef TIME_SIZE
}
/******************************************************************************/

void update(int np,int nd,double pos[],double vel[],double f[],double acc[],double
mass,double dt)
/******************************************************************************/
/*
  Purpose:
    UPDATE updates positions, velocities and accelerations.
  Discussion:
    The time integration is fully parallel.
    A velocity Verlet algorithm is used for the updating.
    x(t+dt) = x(t) + v(t) * dt + 0.5 * a(t) * dt * dt
    v(t+dt) = v(t) + 0.5 * ( a(t) + a(t+dt) ) * dt
    a(t+dt) = f(t) / m
  Licensing:
    This code is distributed under the GNU LGPL license.
  Modified:
    21 November 2007
  Author:
    John Burkardt.
  Parameters:
    Input, int NP, the number of particles.
    Input, int ND, the number of spatial dimensions.
    Input/output, double POS[ND*NP], the positions.
    Input/output, double VEL[ND*NP], the velocities.
    Input, double F[ND*NP], the forces.
```

```
    Input/output, double ACC[ND*NP], the accelerations.
    Input, double MASS, the mass.
    Input, double DT, the time step.
*/
{
  int i;
  int j;
  double rmass;
  rmass = (1.0 / mass);
  for (j = 0; j < np; j++) {
    for (i = 0; i < nd; i++) {
      pos[i + (j * nd)] = ((pos[i + (j * nd)] + (vel[i + (j * nd)] * dt)) + (((0.5 * acc[i
+ (j * nd)]) * dt) * dt));
      vel[i + (j * nd)] = (vel[i + (j * nd)] + ((0.5 * dt) * ((f[i + (j * nd)] * rmass) +
acc[i + (j * nd)])));
      acc[i + (j * nd)] = (f[i + (j * nd)] * rmass);
    }
  }
}
```