Lab 3
Vivian Li, Nick Rovito

(a) Implemented Design

(i) Purpose of the Lab

In this lab, we are looking at how particles move in an Eulerian Grid, where the grid stays fixed but material (i.e. particles) moves through it. Particle simulations are expensive. Discretizing materials as a set of discrete particles, such as heavily deforming solids or fluids, is memory intensive; many materials require a large number of particles to adequately capture their deformation. To save on computational time, it is often necessary to split the domain to several processes in parallel. Here, we complete functions that describe particle motion through domain segments across several processes.

(ii) Particle Physics

The particle physics are determined through discrete computation of Newton's second law:

$$\sum \underline{F}_i = m_i \underline{a}_i$$

where *i* is the index of each particle; $\underline{F}_i$ is the force acting on particle *i*; m_i is the mass of particle *i*.; and $\underline{a}_i$ is the equation of particle *i.* Each particle is assumed to have unit mass. We set the force acting on all particles as $\underline{F} = \ < 0, \ -0.4 >$ .This lab simulates particles moving under the influence of some artificial gravity. We can explicitly model particle velocity using:

$$\underline{v}_{i,n} = \underline{v}_{i,o} + \frac{\underline{F}_i}{m_i}\Delta t$$

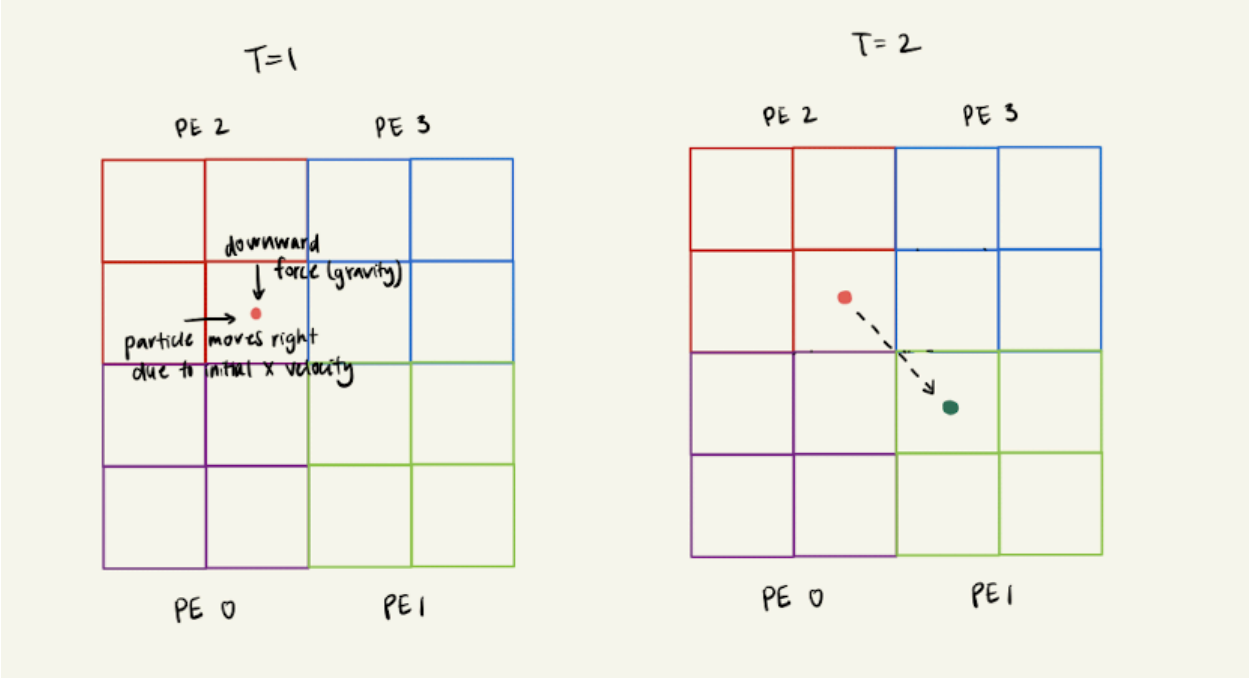
where $\underline{v}_{o,i}$ is the particle velocity at the current time step; $\underline{v}_{n,i}$ is the particle velocity at the new time step; and Δt is the discretized time step.

Similarly, we explicitly model the particle position as:

$$\underline{x}_{n,i} = \underline{x}_{n,i} + \underline{v}_{n,i}\Delta t$$

where $\underline{x}_{o,i}$ is the particle position at the current time step and $\underline{x}_{n,i}$ is the updated position.

Map of PE and ONE particle (image); Particle moves from PE 2 to PE 1 due to the forces acting upon it - essentially one frame of output. PE 2 sends PE 1 the particle's x location, y location, x velocity, and y velocity between time steps 1 and 2. In this example, PE 2 updates its active vector with the new particle.



(iii) MPI Design

Due to the movement of the particles across the mesh, it was necessary for the PEs to communicate with each other to determine which particles should be received by which PE. In order to be memory efficient, we used MPI_Iallreduce to determine the size of the particle array on each PE to accommodate the maximum number of particles that were sent/received. This number was used to initialize all of the contributions that were sent from each PE, which was later populated with the destination PE, x location, y location, x velocities, and y velocities of the particles on this PE. Then, we focused on the Gather operation. We initialized the arrays that would receive the sent buffers using sizeOfGather, which represented the total number of particles that could be sent across all PEs. Then, we used MPI_Iallgather to receive all of the contributions into our gather arrays, so that each PE had the correct information to update their particle array with. Finally, we populated the gather array on each PE using the information we had received from the other PEs. The x location, y location, x velocities, and y velocities of the particles on each array were added to the PTCL object, completing one timestep of our simulation.

(b) Self-Evaluation

This lab was a lot more straightforward for us after all of the lessons we had learned in the last lab. We were able to easily index the PE's neighbors and convert between iPE/jPE/myPE. The MPI reduce and gather functions were also easy to fill in when we knew which arrays we would need to send and receive particles from. However, we ran into a "segmentation error" issue from C++ when running the compiled code because we did not know that we were supposed to send the Cptcl and Gptcl vectors instead of their references in memory. Overall, we enjoyed working together in this lab and creating a cool animation!

Appendix A

ParticleExchange routine

```
void ParticleExchange(VI &ptcl_send_list, VI &ptcl_send_PE, particles &PTCL) {
   MPI_Status status;
   MPI_Request request;

   // (1) Get the max number particles to be sent by any particular processor,
   // and make sure all processors know that number.

   int numToSend = ptcl_send_list.size();
   int maxToSend;

   /*
   int MPI_Iallreduce(
                  const void* send_buffer,
                  void* receive_buffer,
                  int count,
                  MPI_Datatype datatype,
                  MPI_Op operation,
                  MPI_Comm communicator,
                  MPI_Request* request);
   */
   MPI_Iallreduce(&numToSend, &maxToSend, 1, MPI_INT, MPI_MAX, MPI_COMM_WORLD,
                  &request);
   MPI_Wait(&request, &status);

   // (2) Allocate contributions to the upcoming Gather operation.  Here, "C"
   // for "Contribution" to be Gathered

   // For each of these vectors, allocate enough space for the max number of
   // particles sent
   int *Cptcl_PE;
   Cptcl_PE = new int[maxToSend];  // Particles' destination PEs
   double *Cptcl_x;
   Cptcl_x = new double[maxToSend];
   double *Cptcl_y;
   Cptcl_y = new double[maxToSend];
   double *Cptcl_vx;
   Cptcl_vx = new double[maxToSend];
   double *Cptcl_vy;
   Cptcl_vy = new double[maxToSend];
```

```
  // (3) Populate contributions on all processors for the upcoming Gather
  // operation

  for (int i = 0; i < maxToSend; ++i) {
    Cptcl_PE[i] = -1;
    Cptcl_x[i] = 0.;
    Cptcl_y[i] = 0.;
    Cptcl_vx[i] = 0.;
    Cptcl_vy[i] = 0.;
  }

  // (4) Populate with all the particles on this PE.  Note that some/most
  // processors will have left-over space in the C* arrays.

  for (int i = 0; i < ptcl_send_list.size(); ++i) {
    int id = ptcl_send_list[i];
    Cptcl_PE[i] = ptcl_send_PE[i];
    Cptcl_x[i] = PTCL.x[id];
    Cptcl_y[i] = PTCL.y[id];
    Cptcl_vx[i] = PTCL.vx[id];
    Cptcl_vy[i] = PTCL.vy[id];
  }

  // (5) Allocate and initialize the arrays for upcoming Gather operation to
  // PE0.  The sizeOfGather takes
  //     into account the number of processors, like this figure:
  //
  //     |<----------------------------- sizeOfGather
  //     ------------------------------>| | | | |
  //     |<- maxToSend    ->|<- maxToSend    ->|<- maxToSend    ->|<-
  //     maxToSend    ->|
  //
+------------------+------------------+------------------+------------------+
  //             PE0               PE1               PE2               PE3

  int sizeOfGather = maxToSend * numPE;  // per the diagram

  int *Gptcl_PE;
  Gptcl_PE = new int[sizeOfGather];
  double *Gptcl_x;
  Gptcl_x = new double[sizeOfGather];
  double *Gptcl_y;
```

```
Gptcl_y = new double[sizeOfGather];
double *Gptcl_vx;
Gptcl_vx = new double[sizeOfGather];
double *Gptcl_vy;
Gptcl_vy = new double[sizeOfGather];

for (int i = 0; i < sizeOfGather; ++i) {
  Gptcl_PE[i] = -1;
  Gptcl_x[i] = 0.;
  Gptcl_y[i] = 0.;
  Gptcl_vx[i] = 0.;
  Gptcl_vy[i] = 0.;
}

// (6)  Gather "Contributions" ("C" arrays) from all PEs onto all PEs into
// these bigger arrays so all PE will know what particles
//      need to go where.

MPI_Barrier(MPI_COMM_WORLD);

/*
int MPI_Iallgather(
               const void* buffer_send,
               int count_send,
               MPI_Datatype datatype_send,
               void* buffer_recv,
               int count_recv,
               MPI_Datatype datatype_recv,
               MPI_Comm communicator,
               MPI_Request* request);
*/
MPI_Iallgather(Cptcl_PE, maxToSend, MPI_INT, Gptcl_PE, maxToSend, MPI_INT,
               MPI_COMM_WORLD, &request);
MPI_Wait(&request, &status);
MPI_Iallgather(Cptcl_x, maxToSend, MPI_DOUBLE, Gptcl_x, maxToSend,
               MPI_DOUBLE, MPI_COMM_WORLD, &request);
MPI_Wait(&request, &status);
MPI_Iallgather(Cptcl_y, maxToSend, MPI_DOUBLE, Gptcl_y, maxToSend,
               MPI_DOUBLE, MPI_COMM_WORLD, &request);
MPI_Wait(&request, &status);
MPI_Iallgather(Cptcl_vx, maxToSend, MPI_DOUBLE, Gptcl_vx, maxToSend,
               MPI_DOUBLE, MPI_COMM_WORLD, &request);
```

```
  MPI_Wait(&request, &status);
  MPI_Iallgather(Cptcl_vy, maxToSend, MPI_DOUBLE, Gptcl_vy, maxToSend,
                 MPI_DOUBLE, MPI_COMM_WORLD, &request);

  MPI_Barrier(MPI_COMM_WORLD);

  // (7) Put in vector form so they can be added to PTCL.  These arrays are
  // 1-based.

  int Np = 0;
  for (int i = 0; i < sizeOfGather; ++i)
    if (Gptcl_PE[i] == myPE) ++Np;

  VD std_add_x;
  std_add_x.resize(Np + 1);
  VD std_add_y;
  std_add_y.resize(Np + 1);
  VD std_add_vx;
  std_add_vx.resize(Np + 1);
  VD std_add_vy;
  std_add_vy.resize(Np + 1);

  int count = 1;
  for (int i = 0; i < sizeOfGather; ++i)
    if (Gptcl_PE[i] == myPE) {  // these are the particles we want to receive
                                // from the other PEs
      std_add_x[count] = Gptcl_x[i];
      std_add_y[count] = Gptcl_y[i];
      std_add_vx[count] = Gptcl_vx[i];
      std_add_vy[count] = Gptcl_vy[i];
      ++count;
    }

  // Add these particles to the list of particles in the PTCL object
  PTCL.add(std_add_x, std_add_y, std_add_vx, std_add_vy);

  // (8) Free up memory

  if (maxToSend > 0) {
    delete[] Cptcl_PE;
    delete[] Cptcl_x;
    delete[] Cptcl_y;
```

```
    delete[] Cptcl_vx;
    delete[] Cptcl_vy;
  }
  if (sizeOfGather > 0) {
    delete[] Gptcl_PE;
    delete[] Gptcl_x;
    delete[] Gptcl_y;
    delete[] Gptcl_vx;
    delete[] Gptcl_vy;
  }
}
```

Calculating neighbors in GridDecomposition

```
if (iPE > 0 && jPE > 0)
    nei_sw = myPE - nPEx -
             1;  // subtract a row (to move south) and 1 PE (to move west)
  if (iPE < nPEx - 1 && jPE > 0)
    nei_se = myPE - nPEx +
             1;  // subtract a row (to move south) and add 1 PE (to move east)
  if (iPE > 0 && jPE < nPEy - 1)
    nei_nw = myPE + nPEx -
             1;  // add a row (to move north) and subtract 1 PE (to move west)
  if (iPE < nPEx - 1 && jPE < nPEy - 1)
    nei_ne =
        myPE + nPEx + 1;  // add a row (to move north) and 1 PE (to move east)
```

Calculating the new PE in fd.cpp

```
if (PTCL.active[k] == 1) {
      iPEnew = myMPI.iPE;
      jPEnew = myMPI.jPE;

      if (PTCL.x[k] < x0) {  // leaving the left boundary
        PTCL.active[k] = -1;
        iPEnew = myMPI.iPE - 1;
      }
      if (PTCL.x[k] > x1) {  // leaving the right boundary
        PTCL.active[k] = -1;
        iPEnew = myMPI.iPE + 1;
      }
      if (PTCL.y[k] < y0) {  // leaving the bottom boundary
        PTCL.active[k] = -1;
```

```
        jPEnew = myMPI.jPE - 1;
      }
      if (PTCL.y[k] > y1) {  // leaving the top boundary
        PTCL.active[k] = -1;
        jPEnew = myMPI.jPE + 1;
      }
```

```
ptcl_send_PE.push_back(
              iPEnew +
              (jPEnew * myMPI.nPEx));  // calculate which PE it needs
                                       // to go to based on iPE and jPE
```

Appendix B

This is the final timestep of our plot.