University of Colorado Boulder
CSCI 5576: High-Performance Scientific Computing
Lab 04

Leo Beck
Varun Hoskere

Theory and parallel design:

This week, we developed the code that governs the effect and motion of electrostatic particles on a grid of cells - hence the abbreviation esPIC. The idea is to simulate the motion of charged particles across a background mesh. Because particle collisions are ignored, the only equation under consideration is:

$$\nabla E = \frac{\rho}{\varepsilon}$$

where E is the electric field, ρ is the charge density; and ε is the dielectric constant. By rearranging, the above equation can also be written as:

$$E = -\nabla V$$

where V is the scalar electric potential, usually noted as Volts. Further using the Poisson equation, substitution can yield

$$\nabla^2 V = an_i + bV$$

Every ion has an effect on its nearest four nodes on the mesh. Therefore, a weight of the ions charge is assigned to each of the four nodes for each ion, based on the distance between them. This is given by

$$q_p = w_i(\text{PTCL}.Q_p)$$

where Q_p is the charge of the particle and q_p is the weight of the charge associated with the i^{th} nearest node. Dirichlet boundary conditions are set, which means that each side of the square has fixed conditions.

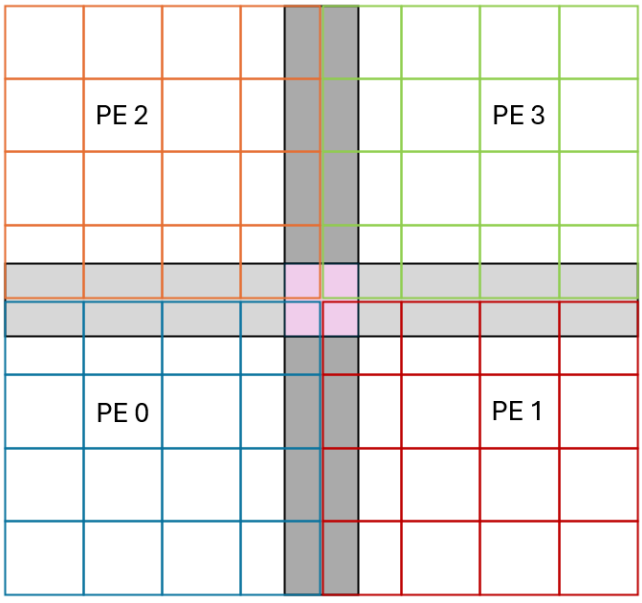

Figure 1: Diagram of the mesh showing 4 total processors

The figure above shows a diagram of the shared sections of the mesh. Each process (at least when using a 2x2 grid of processors) has two edge neighbors, and one catty corner neighbor. So, each process has a north-south edge that needs information from its north-south neighbor (shown in light gray) and an east-west edge with an east-west neighbor (dark gray). Additionally, each processor needs information from each of its neighbors about the center square (pink). To implement this, the code first checks for neighbors in cardinal directions: north, south, east, and west. It then checks for neighbors in ordinal directions based on the cardinal neighbors. From there, the PESum function acting on each processor does a send and receive call to its respective neighbors for the relevant edge (cardinal neighbors) or the single point (ordinal neighbor). Then, it adds the received value(s) to its own value.

For sending values to the cardinal neighbors, we use the phiSend- vectors and similarly for receiving values from the cardinal neighbors, we use the phiRecv- vectors. The catty corner neighbors only need one value to be sent and received, for which we use the nw, ne, sw and se variables for sending and the rec_nw, rec_ne, rec_se and rec_sw variables for receiving.

Verification results:

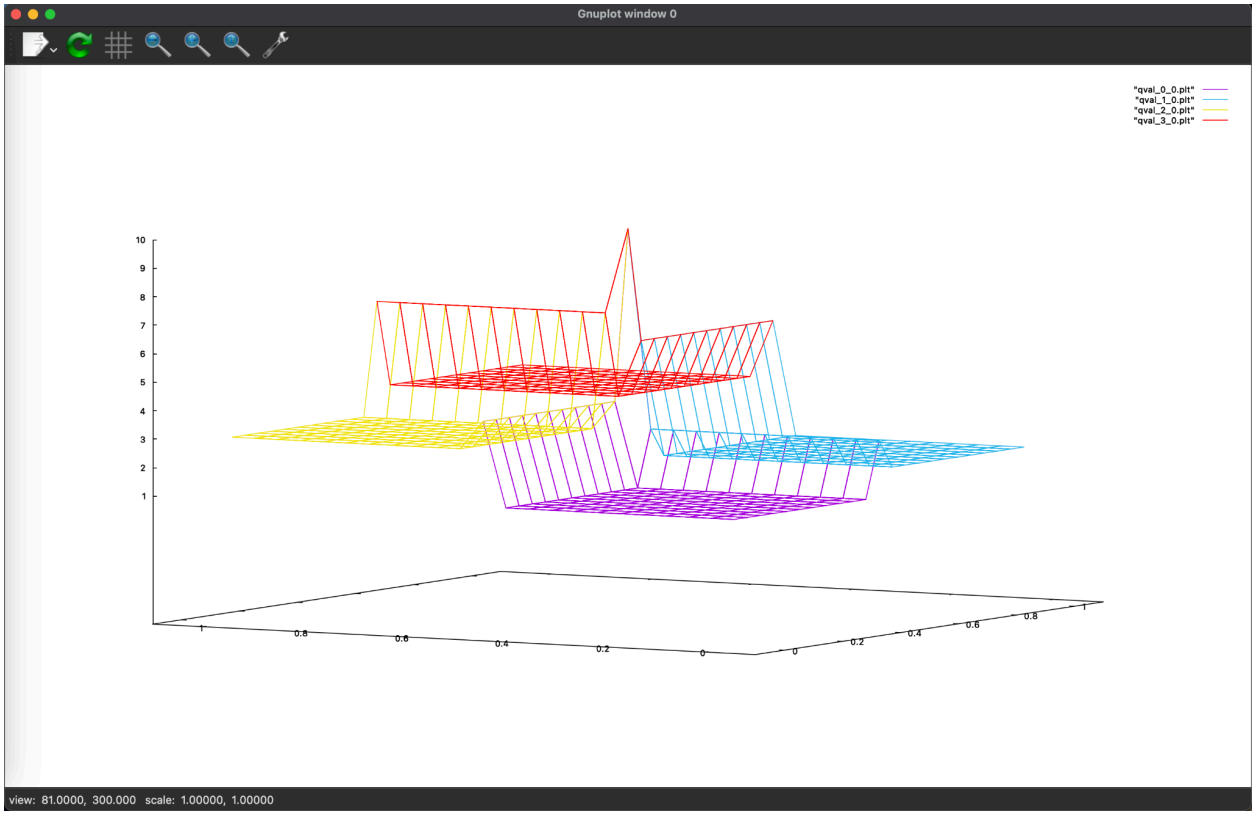

Figure 2: Picture of the functioning validation case

Figure 2 shows the validation case working as expected. The edge between each pair of processors is the sum of their respective values. The center point is the sum of all 4 processors,

equal to 10 (1 + 2 + 3 + 4). After validating, the pc_voltage and pc_qval were plotted (shown below).

Application problem results :

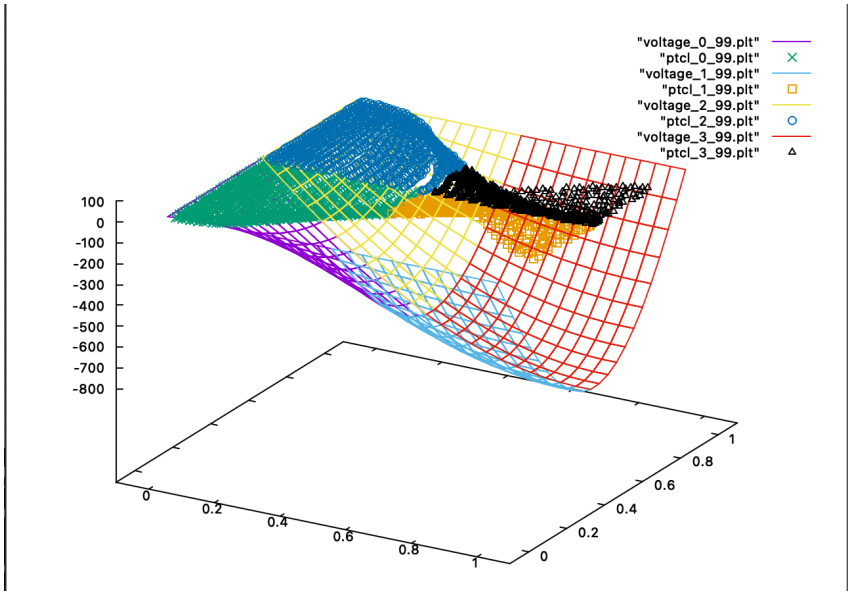

Figure 3: Plot of Pc_voltage

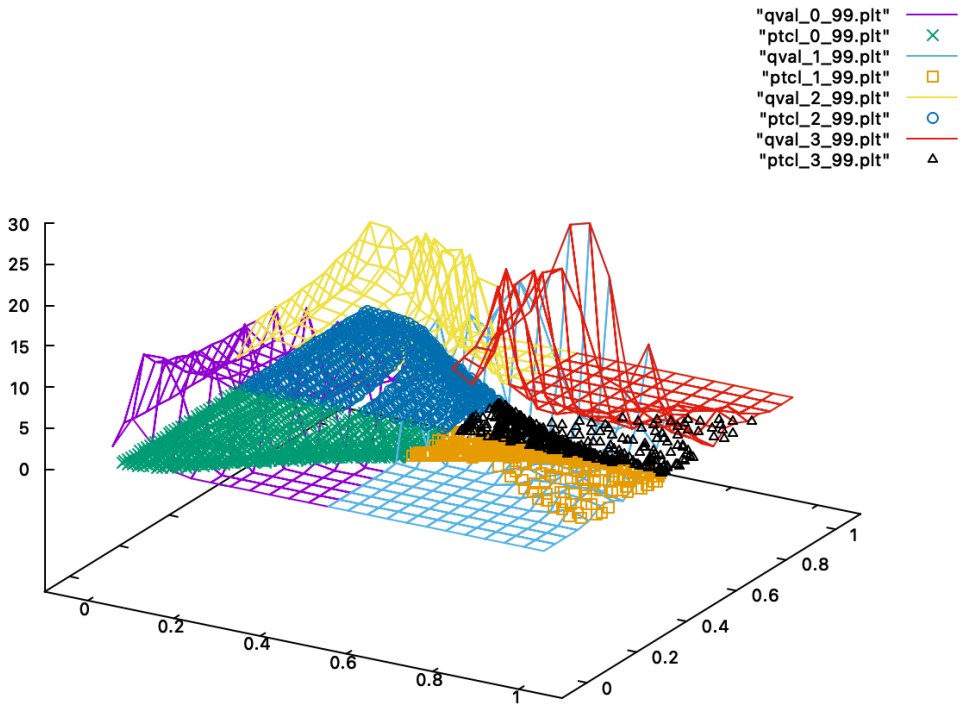

Figure 4: Plot of pc_qval

Self-evaluation:

At first we spent quite a bit of time debating on what the boundaries were, and what communication happens at them. Piece by piece we convinced each other about our thought process and after deliberating for some time, we knew that we had to send buffers of values across the boundaries. We slowly started to see parallels between this week's code and the previous weeks' code - we had buffers to send to the northern, southern, eastern and western neighbors. We re-used the code from last week, making slight modifications. Then we ran the code to see what the output would look like, and we saw that the middle section where the four processors met was not continuous. We then plugged in the values we had previously forgotten - the values for communication for north-eastern, north-western, south-eastern and south-western neighbors. Apart from this, we had another issue - along the east-west direction the boundaries between the processors were discontinuous. This was only along the east-west boundary and not the north-south boundary. This was caused by a small typo in our code as we had missed the '+' in '+='. After fixing this, we had the expected plot and were satisfied with our code.

Appendix A: Code listing of PESum with in-line comments

```
void PEsum( VD &field )
 {
  /*
     find the boundary cells using pid()
     Check iPE, jPE - 'skip' if 0 or nPEx, nPEy (because there are no neighboring processes)
     So, neighboring processes exist at / beyond nodes

     Get relevant PE contributions

     Update field with relevant contributions

     //

     go through the mesh
     if cell is on boundary
     add from neighbours of that cell
        - we know the processor neighbors
        - we also know the cell neighbors
       find the postion of the cell in the field vector

     identifying edge cells: if the i value is nRealx / nRealy then work bothwasys

     if north neighbor exists -

  */



  // if(nei_n >= 0) {

  //   /* determine section of field to grab for edge (top row)*/

  //   iLOOP { phiSend_n[i] = field[pid(i, nRealy)]; }

  // }
  // if(nei_s >= 0) { /* determine section of field to grab for edge (bottom row)*/ iLOOP {}}
  // if(nei_e >= 0) { /* determine section of field to grab for edge (right colum)*/}
  // if(nei_w >= 0) { /* determine section of field to grab for edge (top row)*/}


  // if(nei_ne >= 0){ /* determine section of field to grab for edge (single value)*/}


  iLOOP phiSend_n[i] = field[ pid(    i , nRealy ) ];
```

```
  iLOOP phiSend_s[i] = field[ pid(    i ,    1     ) ];
  jLOOP phiSend_w[j] = field[ pid(    1 ,    j     ) ];
  jLOOP phiSend_e[j] = field[ pid( nRealx ,    j ) ];

  double nw = field[pid(1,nRealy)];
  double ne = field[pid(nRealx, nRealy)];
  double sw = field[pid(1,1)];
  double se = field[pid(nRealx, 1)];

  double rec_nw;
  double rec_ne;
  double rec_sw;
  double rec_se;


  // (1.2) Send them to neighboring PEs

  if ( nei_n >= 0 )  err = MPI_Isend(phiSend_n, countx , MPI_DOUBLE , nei_n , tag ,
MPI_COMM_WORLD , &request );
  if ( nei_s >= 0 )  err = MPI_Isend(phiSend_s, countx , MPI_DOUBLE , nei_s , tag ,
MPI_COMM_WORLD , &request );
  if ( nei_e >= 0 )  err = MPI_Isend(phiSend_e, county , MPI_DOUBLE , nei_e , tag ,
MPI_COMM_WORLD , &request );
  if ( nei_w >= 0 )  err = MPI_Isend(phiSend_w, county , MPI_DOUBLE , nei_w , tag ,
MPI_COMM_WORLD , &request );

  if ( nei_ne >= 0 )  err = MPI_Isend(&ne, 1 , MPI_DOUBLE , nei_ne , tag ,
MPI_COMM_WORLD , &request );
  if ( nei_se >= 0 )  err = MPI_Isend(&se, 1 , MPI_DOUBLE , nei_se , tag ,
MPI_COMM_WORLD , &request );
  if ( nei_nw >= 0 )  err = MPI_Isend(&nw, 1 , MPI_DOUBLE , nei_nw , tag ,
MPI_COMM_WORLD , &request );
  if ( nei_sw >= 0 )  err = MPI_Isend(&sw, 1 , MPI_DOUBLE , nei_sw , tag ,
MPI_COMM_WORLD , &request );



  // (1.3) Receive values from neighobring PEs' physical boundaries.

  if ( nei_n >= 0 ) { err = MPI_Irecv(phiRecv_n, countx , MPI_DOUBLE , nei_n , tag ,
MPI_COMM_WORLD , &request );   MPI_Wait(&request,&status); }
  if ( nei_s >= 0 ) { err = MPI_Irecv(phiRecv_s, countx , MPI_DOUBLE , nei_s , tag ,
MPI_COMM_WORLD , &request );   MPI_Wait(&request,&status); }
  if ( nei_e >= 0 ) { err = MPI_Irecv(phiRecv_e, county , MPI_DOUBLE , nei_e , tag ,
MPI_COMM_WORLD , &request );   MPI_Wait(&request,&status); }
```

```
  if ( nei_w >= 0 ) { err = MPI_Irecv(phiRecv_w, county , MPI_DOUBLE , nei_w , tag ,
MPI_COMM_WORLD , &request );   MPI_Wait(&request,&status); }

  if ( nei_ne >= 0 ) { err = MPI_Irecv(&rec_ne, 1 , MPI_DOUBLE , nei_ne , tag ,
MPI_COMM_WORLD , &request ); MPI_Wait(&request,&status); }
  if ( nei_se >= 0 ) { err = MPI_Irecv(&rec_se, 1 , MPI_DOUBLE , nei_se , tag ,
MPI_COMM_WORLD , &request ); MPI_Wait(&request,&status); }
  if ( nei_nw >= 0 ) { err = MPI_Irecv(&rec_nw, 1 , MPI_DOUBLE , nei_nw , tag ,
MPI_COMM_WORLD , &request ); MPI_Wait(&request,&status); }
  if ( nei_sw >= 0 ) { err = MPI_Irecv(&rec_sw, 1 , MPI_DOUBLE , nei_sw , tag ,
MPI_COMM_WORLD , &request ); MPI_Wait(&request,&status); }

  iLOOP field[pid(i,nRealy)] += phiRecv_n[i];
  iLOOP field[pid(i,1)] += phiRecv_s[i];
  jLOOP field[pid(nRealx, j)] += phiRecv_e[j];
  jLOOP field[pid(1, j)] += phiRecv_w[j];

  field[pid(1,nRealy)] += rec_nw;
  field[pid(nRealx, nRealy)] += rec_ne;
  field[pid(1,1)] += rec_sw;
  field[pid(nRealx, 1)] += rec_se;

 }
```