Math 471 Project 2 NBody parallel

Erik Erhardt

October 26, 2005

1. Problem Description. *Kelvin-Helmholtz Instability*

The Kelvin-Helmholtz (KH) instability results from velocity shears between two volumes (or areas in 2D). Any time there is a non-zero curvature, the flow of one fluid around another will lead to a slight centrifugal force. In lecture, under certain uniformity conditions, the problem can be reduced to the n-body problem concentrated at the boundary.

Our initial condition is given in the first plot in Figure 2 on page 3, where the top layer is moving to the left and the bottom layer to the right. Our goal is to write two parallel implementations to solve this system over time.

2. Scalability. A scalability plot is presented in Figure 1 on page 2. In reference to the $y = x$ line, both the All-gather and Ring methods are nearly perfectly scalable. On Azul, we see that beyond 10 processes, the scalability begins to decrease slightly, and at 28 processes we observe the ring method has diverged more than the all-gather method from perfect scalability.

An interesting result occured, and I hesitate to even bring it up. When I ran my jobs for 20 time steps, the ring method performed in nearly 1/22 the time that all-gather did. However, when I run it to the end, the times are roughly the same. Optimistically, I want to say I was performing at 0.192 sec per timestep, but truely, I think I was just under 4 seconds per timestep.

```
Using NBODY=5600, iter=20, h=0.01
                              RING METHOD                 |             ALL-GATHER
Nl=5600 nprocs= 1, (confused about sending) appr=5.217492  |  time=2187.967665, time_per_step=109.398383
Nl=2800 nprocs= 2, time=52.174929, time_per_step=2.608746  |  time=1093.055366, time_per_step=54.652768
Nl=1400 nprocs= 4, time=26.205138, time_per_step=1.310257  |  time=548.809325,  time_per_step=27.440466
Nl=1120 nprocs= 5, time=20.986650, time_per_step=1.049333  |  time=440.542995,  time_per_step=22.027150
Nl=800  nprocs= 7, time=15.012021, time_per_step=0.750601  |  time=315.207031,  time_per_step=15.760352
Nl=700  nprocs= 8, time=13.198580, time_per_step=0.659929  |  time=275.900288,  time_per_step=13.795014
Nl=560  nprocs=10, time=10.557371, time_per_step=0.527869  |  time=220.755420,  time_per_step=11.037771
Nl=400  nprocs=14, time=7.566500,  time_per_step=0.378325  |  time=158.085933,  time_per_step=7.904297
Nl=350  nprocs=16, time=6.637210,  time_per_step=0.331860  |  time=138.815771,  time_per_step=6.940789
Nl=280  nprocs=20, time=5.367745,  time_per_step=0.268387  |  time=111.133442,  time_per_step=5.556672
Nl=224  nprocs=25, time=4.285132,  time_per_step=0.214257  |  time=89.062379,   time_per_step=4.453119
Nl=200  nprocs=28, time=3.840853,  time_per_step=0.192043  |  time=79.649278,   time_per_step=3.982464

time_per_step for RING METHOD to t=0.6 (h=0.0007) is 3.827700
```

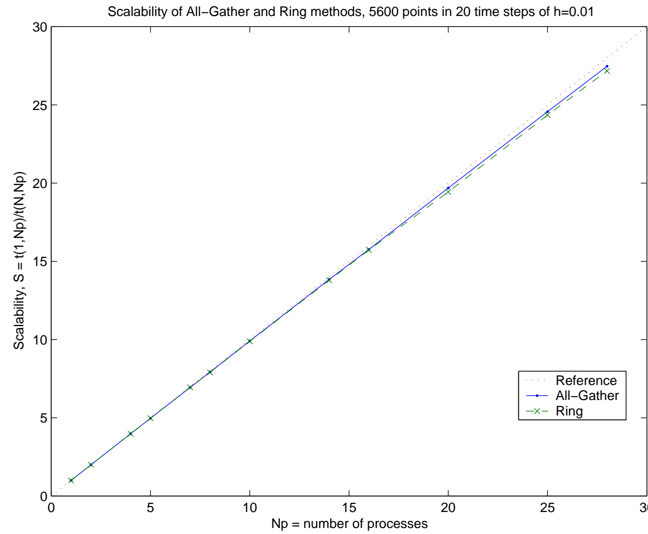

Figure 1: Scalability plot.

3. Best wall-clock time per timestep. Using $h = 0.01$, All-gather performs with 28 processors at roughly 3.79767 sec per timestep.

4. Necessary time step. I reduced the timestep-size down to $h = 0.0007$, but $h = 0.001$ seemed sufficient.

5. Plots and Relative error. Five plots are presented in Figure 2 on page 3. The first four plots show the evolution of the boundary at $t = 0.0, 0.2, 0.4, 0.6$ using $N_p = 5600$ and $h = 0.001$. The fifth plot shows a zoom of the left vortex at $t = 0.6$ to show it has converged.

The Table 1 on page 2 shows the relative error in the energy for $h = 0.001$ and $h = 0.0007$. Also are shown the total time for the run in seconds and the average time per step. The error is very little, and the rate of error decreases slightly in time. The plot in Figure 3 on page 4 is a plot of this.

$h = 0.001$		$h = 0.0007$	
time=2282	per step=3.79797	time=3262	per step=3.79767
t	H rel error	t	H rel error
0.000000	0.000000	0.000000	0.000000
0.200000	-0.000012	0.200200	-0.000006
0.400000	-0.000021	0.400400	-0.000009
0.600000	-0.000027	0.600600	-0.000011

Table 1: Relative error in the energy for $h = 0.001$ and $h = 0.0007$

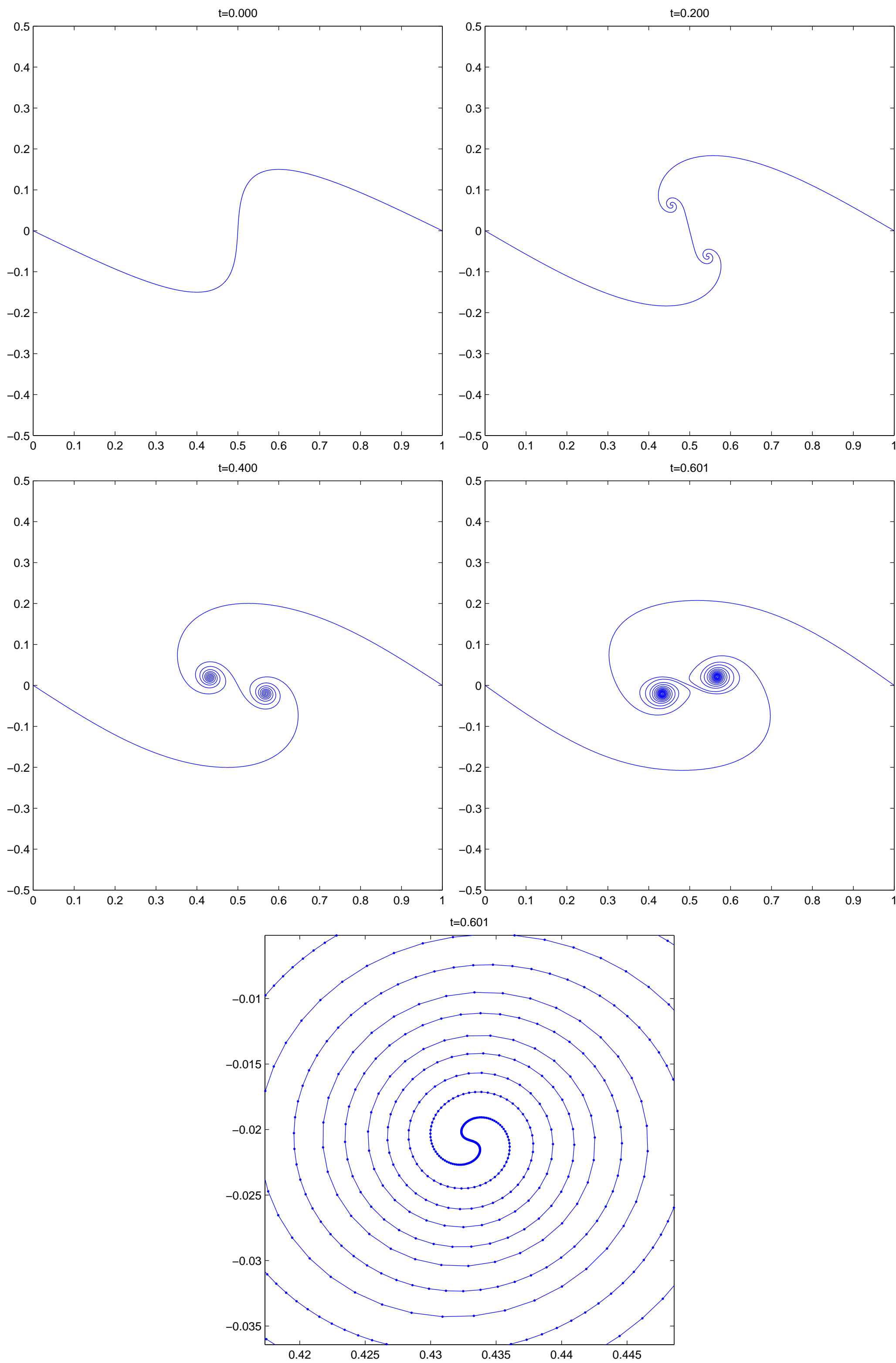

Figure 2: Plots of vorticies at $t = 0.0, 0.2, 0.4, 0.6$ and zoom of left vortex at $t = 0.6$, using $N_p = 5600$ and $h = 0.001$.

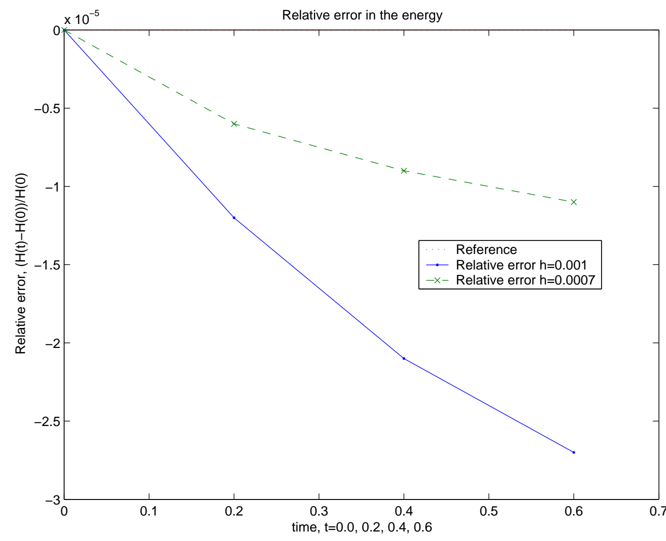

Figure 3: Relative error in energy $t = 0.0, 0.2, 0.4, 0.6$ using $N_p = 5600$ with $h = 0.001$ and $h = 0.0007$.

Appendix

Matlab and C code

```
%Scalability
%np  timestepAG      timestepR
x=[
 1  109.398383 5.217492
 2  54.652768  2.608746
 4  27.440466  1.310257
 5  22.027150  1.049333
 7  15.760352  0.750601
 8  13.795014  0.659929
10  11.037771  0.527869
14  7.904297   0.378325
16  6.940789   0.331860
20  5.556672   0.268387
25  4.453119   0.214257
28  3.982464   0.192043
]'
s=[x(2,1)./x(2,:);x(3,1)./x(3,:)];

plot([0,30],[0,30],'r:', x(1,:),s(1,:),'.-', x(1,:),s(2,:),'x--');
legend('Reference','All-Gather','Ring',0);
title('Scalability of All-Gather and Ring methods, 5600 points in 20 time steps of h=0.01');
xlabel('Np = number of processes');
ylabel('Scalability, S = t(1,Np)/t(N,Np)');

% relative error
x=[
0.000000 0  0.000000  0.000000
0.200000 0 -0.000012 -0.000006
0.400000 0 -0.000021 -0.000009
0.600000 0 -0.000027 -0.000011
]'
plot(x(1,:),x(2,:),'r:', x(1,:),x(3,:),'.-', x(1,:),x(4,:),'x--')
legend('Reference','Relative error h=0.001','Relative error h=0.0007',0);
title('Relative error in the energy');
xlabel('time, t=0.0, 0.2, 0.4, 0.6');
ylabel('Relative error, (H(t)-H(0))/H(0)');


/******************************************************************************
Erik Erhardt
Math 471
Testing MPI_Isend and MPI_Irecv

on frontend:
  make nbody
  mpirun -np 4 ./nbody

on backend (many nodes):
  qsub -I -l nodes=7,walltime=900
  mpirun -np 28 -machinefile $PBS_NODEFILE ./nbody
  qsub -I -l nodes=3,walltime=900
  mpirun -np 12 -machinefile $PBS_NODEFILE ./nbody
******************************************************************************/
/******************************************************************************
Times:

h=0.001
t=0.000000, H_rel_error= 0.000000
t=0.200000, H_rel_error=-0.000012
t=0.400000, H_rel_error=-0.000021
t=0.600000, H_rel_error=-0.000027
time=2282.582404, time_per_step=3.797974

h=0.0007
t=0.000000, H_rel_error= 0.000000
t=0.200200, H_rel_error=-0.000006
t=0.400400, H_rel_error=-0.000009
t=0.600600, H_rel_error=-0.000011
time=3262.199582, time_per_step=3.797671

******************************************************************************/

#include <mpi.h>
#include <math.h>

   // 400 or 5600
#define NBODY 5600

double PI;
double t, x[NBODY*2];
double xprime[NBODY*2], x_rhs_temp[NBODY*2];
double dx_r[NBODY], dy_r[NBODY];        // ring
double h, t0, tfinal, time_per_step;
int n;
int sw_method, sw_output, sw_scalability, sw_scale_iter=0;
double delta;
int Nl, tag=0;                        // Nl=number of points per proc
```

```
int lower, upper;
int rank, nprocs;

int main(int argc, char *argv[]){
   int i, j;
   double time1, time2, timetotal;         // time the run
   int junk;
   MPI_Init(&argc,&argv);                  // initialize MPI
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);   // my rank number
   MPI_Comm_size(MPI_COMM_WORLD, &nprocs); // number of processes
   Nl=NBODY/nprocs;                        // number per process
   if (rank==0) printf("NBODY=%d, Nl=%d\n", NBODY, Nl);
   junk = init();  /*******/               // initialize values
   if ( sw_scalability==1) {
      if (rank==0) printf("Scalability time trial using nprocs=%d, sw_scale_iter=%d\n", nprocs, sw_scale_iter);
   }
   time1 = MPI_Wtime();                    // first time t1
   if (sw_method==0){ junk = rk2_fluid(); } // allgather time step
   if (sw_method==1){ junk = rk2_ring_fluid(); } // ring time step
   time2 = MPI_Wtime();                       // first time t1
   timetotal = time2 - time1;                        // total time is difference
   if (sw_scale_iter > 0) n=sw_scale_iter;
   time_per_step = timetotal/n;
   if (rank==0) printf("time=%f, time_per_step=%f\n", timetotal, time_per_step);
   MPI_Finalize();
}

/* init *****************************************/
int init(){
   double Gamma, alpha;
   int j;
   lower = rank*Nl;
   upper = (rank+1)*Nl;
   // printf("rank=%d, lower=%d, upper=%d\n",rank,lower,upper);
   PI=4*atan( (double)1 );                 // known value of pi
   sw_method = 1;                      // 0 all_gather, 1 ring
   sw_output = 1;                      // 0 no output, 1 output to screen and file
   sw_scalability = 0;
   if ( sw_scalability==1) {
      sw_output = 0;
      sw_scale_iter = 20;
   }
   //Warm-up init NBODY=400
   /*
   h=0.1; t0=0.0; tfinal=4.0;
   n = ceil((tfinal-t0)/h)+1;       // number of timesteps
   delta=0.25;   alpha = 0.01;
   */
   // Project init  NBODY=5600
   h=0.0007; t0=0.0; tfinal=0.6;
   n = ceil((tfinal-t0)/h)+1;       // number of timesteps
   delta=0.04;   alpha = 0.15;
//   for(j=0; j<NBODY; j++){
   for(j=lower; j<upper; j++){
      Gamma=(double) j/NBODY;
      x[NBODY+j] = -alpha*sin(2*PI*Gamma);  // initial positions y direction
      x[j]       = Gamma-x[NBODY+j];    // initial positions x direction
      // printf("j=%d, n=%d, NBODY=%d, alpha=%f, Gamma=%f, 2*PI*Gamma=%e, x[j]=%e, x[NBODY+j]=%e\n",j,n,NBODY,alpha,Gam
   }
   t = t0;
   return 0;
}

/* rk2_fluid allgather sw_method *****************************************/
// rk2 -- Runge-Kutta method
int rk2_fluid(){
   int ierr1;
   double x_indy[2*Nl], x_rhs_allgather[2*NBODY], x_rhs_allgather_sorted[2*NBODY];
   double x_output[NBODY], y_output[NBODY];
   double H_sum, H0, H_rel_error, PI2, delta2, H_final;
   int i, j, j1, j2, k;
   int junk;
```

```
if(sw_output==1){
if (rank==0) printf("n=%4d, tfinal=%f\n   i      t\n", n, tfinal);}
/* MAIN LOOP *************************/
for ( i=0; i<n; i++) {
   if(sw_output==1){
   if (rank==0) printf("%4d %2.3f \n", i, t);}
   /* send results to all processes ***************/
   for (j=lower; j<upper; j++) {
      x_indy[j-lower] = x[j];
      x_indy[Nl+j-lower] = x[NBODY+j];
   }
   ierr1 = MPI_Allgather(x_indy, 2*Nl, MPI_DOUBLE, x_rhs_allgather, 2*Nl, MPI_DOUBLE, MPI_COMM_WORLD);
      if (ierr1 != MPI_SUCCESS) printf("MPI_Allgather ERROR status %d.\n", ierr1);
   /* unscramble x and y components (allgather has x1,y1,x2,y2,..., want x1,x2...,y1,y2... */
   for (j1=0, k=0; j1<nprocs; j1++){
      for (j2=0; j2<Nl; j2++, k++){
         x[k]       = x_rhs_allgather[2*Nl*j1+j2];
         x[NBODY+k] = x_rhs_allgather[2*Nl*j1+j2+Nl];
      }
   }
   //if (rank==0) {for(j=0;j<NBODY;j++){printf("Org %d\t%f\t%f\n",j,x[j],x[NBODY+j]);}}
   /* Occationally, output positions to a file ************/
   if(sw_output==1){
   if (
       (i == 0) ||
       (t >= 0.2 && t < 0.2+h) ||
       (t >= 0.4 && t < 0.4+h) ||
       (i == n-1)
      ) {
      // H energy **************************
      PI2=pow(PI,2);delta2=pow(delta,2);
      H_sum=0; H_final=0;
      for (j1=lower; j1<upper; j1++){
         for (j2=0; j2<NBODY; j2++){
            H_sum = H_sum + log(cosh(2*PI*(x[j1+NBODY]-x[j2+NBODY]))-cos(2*PI*(x[j1]-x[j2]))+delta2);
         }
      }
      MPI_Reduce(&H_sum, &H_final, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
      if (rank==0) H_sum=(H_sum - Nl*log(delta2))/(-8*PI*pow(NBODY,2));
      if (rank==0) H_final=(H_final - NBODY*log(delta2))/(-8*PI*pow(NBODY,2));
      //if (rank==0) printf("i=%d, H_sum=%f, H_final=%f\n", i, H_sum, H_final);
      if(i==0) H0=H_final;
      H_rel_error = (H_final-H0)/H0;
      if (rank==0) printf("t=%f, H_rel_error=%f\n", t, H_rel_error);
      // print positions to file
      for (j=0; j<NBODY; j++){ x_output[j] = x_indy[j]; y_output[j] = x_indy[Nl+j]; }
      junk = output(Nl, NBODY, x_output, y_output, t, H_final);
   }
   }

   // original x into temp vector for rhs
   for (j=0; j<NBODY; j++) {
      x_rhs_temp[j]       = x[j];
      x_rhs_temp[NBODY+j] = x[NBODY+j];
   }
   //if (rank==0) {for(j=0;j<NBODY;j++){printf("0th %d\t%f\t%f\n",j,x_rhs_temp[j],x_rhs_temp[NBODY+j]);}}
   /* k1 ******************************************************************************************/
   junk = n_body_rhs_fluid(1,i);
   /* send results to all processes ***************/
   for (j=lower; j<upper; j++) {
      x_indy[j-lower]    = x_rhs_temp[j];
      x_indy[Nl+j-lower] = x_rhs_temp[NBODY+j];
   }
   ierr1 = MPI_Allgather(x_indy, 2*Nl, MPI_DOUBLE, x_rhs_allgather, 2*Nl, MPI_DOUBLE, MPI_COMM_WORLD);
      if (ierr1 != MPI_SUCCESS) printf("MPI_Allgather ERROR status %d.\n", ierr1);
   /* unscramble x and y components (allgather has x1,y1,x2,y2,..., want x1,x2...,y1,y2... */
   for (j1=0, k=0; j1<nprocs; j1++){
      for (j2=0; j2<Nl; j2++, k++){
         x_rhs_allgather_sorted[k]       = x_rhs_allgather[2*Nl*j1+j2];
         x_rhs_allgather_sorted[NBODY+k] = x_rhs_allgather[2*Nl*j1+j2+Nl];
      }
   }
   for ( j=0; j<2*NBODY; j++) { x_rhs_temp[j]=x_rhs_allgather_sorted[j]; }  // put into xprime
```

```
      //if (rank==0) {for(j=0;j<NBODY;j++){printf("1st %d\t%f\t%f\n",j,x_rhs_temp[j],x_rhs_temp[NBODY+j]);}}
      /* k2 ******************************************************************************************/
      junk = n_body_rhs_fluid(2,i);
      t = t + h;
      if ( sw_scalability==1 && i >= 20) {i=n+1;}
   }  // finish i loop
   if(sw_output==1){
   if(i==1) printf("i=%d\n",i);}
   /* send results to all processes ***************/
   for (j=lower; j<upper; j++) {
      x_indy[j-lower] = x[j];
      x_indy[Nl+j-lower] = x[NBODY+j];
   }
   ierr1 = MPI_Allgather(x_indy, 2*Nl, MPI_DOUBLE, x_rhs_allgather, 2*Nl, MPI_DOUBLE, MPI_COMM_WORLD);
      if (ierr1 != MPI_SUCCESS) printf("MPI_Allgather ERROR status %d.\n", ierr1);
   /* unscramble x and y components (allgather has x1,y1,x2,y2,..., want x1,x2...,y1,y2... */
   for (j1=0, k=0; j1<nprocs; j1++){
      for (j2=0; j2<Nl; j2++, k++){
         x[k]       = x_rhs_allgather[2*Nl*j1+j2];
         x[NBODY+k] = x_rhs_allgather[2*Nl*j1+j2+Nl];
      }
   }
   return 0;
}

/* n_body_rhs_fluid allgather sw_method *****************************************/
int n_body_rhs_fluid(int sw_k1k2, int i){
   double x_rhs[NBODY], y_rhs[NBODY];
   double dx[NBODY], dy[NBODY];
   double pi2, delta2, sinhy, sinx, denom, xd, yd;
   int j, k;
   memset(dx,0,NBODY*sizeof(dx[1]));
   memset(dy,0,NBODY*sizeof(dy[1]));
   for (j=0; j<NBODY; j++) {
      x_rhs[j] = x_rhs_temp[j];
      y_rhs[j] = x_rhs_temp[NBODY+j];
   }
   pi2 = 2*PI;
   delta2 = pow(delta,2);
   for (j=lower; j<upper; j++) {
      for (k=0; k<NBODY; k++) {
         xd = x_rhs[j]-x_rhs[k];
         yd = y_rhs[j]-y_rhs[k];
         sinhy = sinh(pi2*yd);
         sinx = sin(pi2*xd);
         denom = cosh(pi2*yd)-cos(pi2*xd)+delta2;
         dx[j] = dx[j] + sinhy/denom;
         dy[j] = dy[j] + sinx/denom;
      }
      dx[j]=dx[j]/(-2*NBODY);
      dy[j]=dy[j]/( 2*NBODY);
   }
   // pack dxdt and dvdt into xprime array:
   /* k1 or k2 */
   if (sw_k1k2==1){
      for (j=lower; j<upper; j++) {
         x_rhs_temp[j]       = x[j]       + (h*dx[j])/2;
         x_rhs_temp[NBODY+j] = x[NBODY+j] + (h*dy[j])/2;
      }
   }
   if (sw_k1k2==2) {
      for (j=lower; j<upper; j++) {
         x[j]       = x[j]       + (h*dx[j]);
         x[NBODY+j] = x[NBODY+j] + (h*dy[j]);
      }
   }
   return 0;
}

/******************************************************************************/
/******************************************************************************/
/* rk2_ring_fluid ring sw_method *****************************************/
```

```
// rk2 -- Runge-Kutta method
int rk2_ring_fluid(){
   int ierr, ierr1, ierr2, ierr3;
   double x_indy[2*Nl], x_loc[2*Nl], x_rem[2*Nl], x_rem_tosend[2*Nl];
   double x_output[Nl], y_output[Nl];
   double H_sum, PI2, delta2, H_final;
   int i, j, j1, j2, k, it;
   int junk;
   int sendto, recvfrom;
   MPI_Status status;
   MPI_Request rreq, sreq;

   if(sw_output==1){
   if (rank==0) printf("n=%4d, tfinal=%f\n   i     t\n", n, tfinal);}
   sendto   = rank+1; if(sendto>nprocs-1) sendto=0;      // send to next
   recvfrom = rank-1; if(recvfrom<0) recvfrom=nprocs-1;  // recv from previous

   // printf("rank=%d, sendto=%d, recvfrom=%d\n", rank, sendto, recvfrom);

   // for(j=0;j<2*NBODY;j++){x[j]=rank;} //testing

   for (j=lower; j<upper; j++) {x_indy[j-lower] = x[j]; x_indy[Nl+j-lower] = x[NBODY+j];}
   for (j=0; j<2*Nl; j++) {x[j] = x_indy[j]; x_loc[j]=x_indy[j];}  // reposition x values // local values

   /* MAIN LOOP *************************/
   for ( i=0; i<n; i++) {
      if(sw_output==1){
      if (rank==0) printf("%4d %2.3f \n", i, t);}

      /* Occationally, output positions to a file ************/
      if(sw_output==1){
      if (
          (i == 0) ||
          (t >= 0.2 && t < 0.2+h) ||
          (t >= 0.4 && t < 0.4+h) ||
          (i == n-1)
         ) {
//       // H energy *************************
//       PI2=pow(PI,2);delta2=pow(delta,2);
//       H_sum=0; H_final=0;
//       for (j1=0; j1<Nl; j1++){
//          for (j2=0; j2<Nl; j2++){
//             H_sum = H_sum + log(cosh(2*PI*(x[j1+Nl]-x[j2+Nl]))-cos(2*PI*(x[j1]-x[j2]))+delta2);
//          }
//       }
//       MPI_Reduce(&H_sum, &H_final, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
//       if (rank==0) H_sum=(H_sum - Nl*log(delta2))/(-8*PI*pow(NBODY,2));
//       if (rank==0) H_final=(H_final - NBODY*log(delta2))/(-8*PI*pow(NBODY,2));
//       //if (rank==0) printf("i=%d, H_sum=%f, H_final=%f\n", i, H_sum, H_final);

         // print positions to file
         H_final=0;
         for (j=0; j<Nl; j++){ x_output[j] = x[j]; y_output[j] = x[Nl+j]; }
         junk = output(Nl, NBODY, x_output, y_output, t, H_final);
      }
//      OUTPUT EVERYTIME
//      for (j=0; j<Nl; j++){ x_output[j] = x[j]; y_output[j] = x[Nl+j]; }
//      junk = output(Nl, NBODY, x_output, y_output, t, H_final);
      }

      /* k1 ***********************************************************************************************/
      for (j=0; j<2*Nl; j++) {x_rem[j]=x_loc[j];}     // local values for first remote send
      for (j=0; j<Nl; j++) {
         x_rhs_temp[j]         = x_loc[j];      // x_loc
         x_rhs_temp[Nl+j]      = x_rem[j];      // x_rem
         x_rhs_temp[2*Nl+j]    = x_loc[Nl+j];   // y_loc
         x_rhs_temp[2*Nl+Nl+j] = x_rem[Nl+j];   // y_rem
      }

      junk = n_body_rhs_ring_fluid(1,i,0);
      for(it=1; it<nprocs; it++){
      //   if(it>0){
            // THIS CODE IS THE RING METHOD
            for (j=0; j<2*Nl; j++) {x_rem_tosend[j]=x_rem[j];}  // local values to send
            ierr1 = MPI_Irecv(&x_rem,        2*Nl, MPI_DOUBLE, recvfrom, tag, MPI_COMM_WORLD, &rreq);if (ierr1 != MPI_S
            ierr2 = MPI_Isend(&x_rem_tosend, 2*Nl, MPI_DOUBLE, sendto,   tag, MPI_COMM_WORLD, &sreq);if (ierr2 != MPI_S
            ierr3 = MPI_Wait(&rreq, &status);                                                        if (ierr3 != MPI_SUC
      //   }
         for (j=0; j<Nl; j++) {
            x_rhs_temp[j]         = x_loc[j];      // x_loc
            x_rhs_temp[Nl+j]      = x_rem[j];      // x_rem
            x_rhs_temp[2*Nl+j]    = x_loc[Nl+j];   // y_loc
```

```
            x_rhs_temp[2*Nl+Nl+j] = x_rem[Nl+j];   // y_rem
         }
         junk = n_body_rhs_ring_fluid(1,i,it);
         ierr3 = MPI_Wait(&sreq, &status);                                                      if (ierr3 != MPI_SUCCES
      }
      //exit(0);//testing
      /* Update x_loc values ***************/
      for (j=0; j<2*Nl; j++) {x_loc[j] = x_rhs_temp[j];}
      /* k2 ****************************************************************************************/
      for (j=0; j<2*Nl; j++) {x_rem[j]=x_loc[j];}     // local values for first remote send
      for (j=0; j<Nl; j++) {
         x_rhs_temp[j]         = x_loc[j];      // x_loc
         x_rhs_temp[Nl+j]      = x_rem[j];      // x_rem
         x_rhs_temp[2*Nl+j]    = x_loc[Nl+j];   // y_loc
         x_rhs_temp[2*Nl+Nl+j] = x_rem[Nl+j];   // y_rem
      }
      junk = n_body_rhs_ring_fluid(2,i,0);
      for(it=1; it<nprocs; it++){
         //if(it>0){
            // THIS CODE IS THE RING METHOD
            for (j=0; j<2*Nl; j++) {x_rem_tosend[j]=x_rem[j];}  // local values to send
            ierr1 = MPI_Irecv(&x_rem,        2*Nl, MPI_DOUBLE, recvfrom, tag, MPI_COMM_WORLD, &rreq);if (ierr1 != MPI_S
            ierr2 = MPI_Isend(&x_rem_tosend, 2*Nl, MPI_DOUBLE, sendto,   tag, MPI_COMM_WORLD, &sreq);if (ierr2 != MPI_S
            ierr3 = MPI_Wait(&rreq, &status);                                                      if (ierr3 != MPI_SUC
         //}
         for (j=0; j<Nl; j++) {
            x_rhs_temp[j]         = x_loc[j];      // x_loc
            x_rhs_temp[Nl+j]      = x_rem[j];      // x_rem
            x_rhs_temp[2*Nl+j]    = x_loc[Nl+j];   // y_loc
            x_rhs_temp[2*Nl+Nl+j] = x_rem[Nl+j];   // y_rem
         }
         junk = n_body_rhs_ring_fluid(2,i,it);
         ierr3 = MPI_Wait(&sreq, &status);                                                      if (ierr3 != MPI_SUCCES
      }
      /* Update x_loc values ***************/
      for (j=0; j<2*Nl; j++) { x_loc[j] = x[j];}
      t = t + h;
      if ( sw_scalability==1 && sw_scale_iter >= 20) {i=n+1;}
   }  // finish i loop
   //for (j=0; j<Nl; j++) { printf("%f %f\n",x[j], x[Nl+j]);}  // plot final points

   return 0;
}

/* n_body_rhs_ring_fluid ring sw_method ****************************************/
int n_body_rhs_ring_fluid(int sw_k1k2, int i, int it){
   double x_rhs[NBODY], y_rhs[NBODY];
   double dx_r_t[NBODY], dy_r_t[NBODY];
   double pi2, delta2, sinhy, sinx, denom, xd, yd;
   int j, k;
   if (it==0) {
      memset(dx_r,0,NBODY*sizeof(dx_r[1]));
      memset(dy_r,0,NBODY*sizeof(dy_r[1]));
   }
   memset(dx_r_t,0,NBODY*sizeof(dx_r_t[1]));
   memset(dy_r_t,0,NBODY*sizeof(dy_r_t[1]));
   for (j=0; j<2*Nl; j++) {
      x_rhs[j] = x_rhs_temp[j];
      y_rhs[j] = x_rhs_temp[2*Nl+j];
    //printf("RHSr=%d, it=%d, j=%d, %f\t%f\n", rank, it, j,
    //      x_rhs[j],
    //      y_rhs[j]
    //      );                                                        //testing
   }
   pi2 = 2*PI;
   delta2 = pow(delta,2);
   for (j=0; j<Nl; j++) {
      for (k=Nl; k<2*Nl; k++) {
         xd = x_rhs[j]-x_rhs[k];
         yd = y_rhs[j]-y_rhs[k];
```

```
         sinhy = sinh(pi2*yd);
         sinx = sin(pi2*xd);
         denom = cosh(pi2*yd)-cos(pi2*xd)+delta2;
         dx_r_t[j] = dx_r_t[j] + sinhy/denom;
         dy_r_t[j] = dy_r_t[j] + sinx/denom;
      }
      dx_r[j]=dx_r[j]+dx_r_t[j]/(-2*NBODY);
      dy_r[j]=dy_r[j]+dy_r_t[j]/( 2*NBODY);
   }
   //if(rank==0) printf("n_body_rhs i=%d, n=%d, it=%d, nprocs=%d \n", i, n, it, nprocs);
   if(it==nprocs-1){
      // pack dx_rdt and dvdt into xprime array:
      if (sw_k1k2==1){
         for (j=0; j<Nl; j++) {
            x_rhs_temp[j]    = x[j]    + (h*dx_r[j])/2;
            x_rhs_temp[Nl+j] = x[Nl+j] + (h*dy_r[j])/2;
         }
      }
      if (sw_k1k2==2) {
         for (j=0; j<Nl; j++) {
            x[j]    = x[j]    + (h*dx_r[j]);
            x[Nl+j] = x[Nl+j] + (h*dy_r[j]);
         }
      }
   }
   return 0;
}

/*
!  output data from M471 point vortex N-body calculation
!
!  input:   n   number of points on each process
!           ng  total number of points
!           x(n)  array containing x-coordinates
!           y(n)  array containing y-coordinates
!           time
!           H     Hamiltonian
!
!  This routine will use MPI_Gather to collect all the coordinates
!  on rank==0, and then use MPI-IO to write a file, "kh.out"
!
!  Call this routine once for each snapshot you wish to save to
!  that it can be plotted in Matlab.
!
!  The first time you call this routine, it will create the file
!  and write (as IEEE double precision data) the input data:
!  n,time,H,x(1:ng) and y(1:ng).
!
!  subsequent calls will append to the same file
!
*/
int output(int n, int ng, double *x, double *y, double time, double H){
   static int firstcall=1;
   int amode,rank;
   char *fname="kh.out";
   double xbig[ng];
   MPI_File fh;

   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   //if (rank==0) printf("output routine\n");
   if (firstcall) {
       amode = MPI_MODE_WRONLY | MPI_MODE_CREATE;
       MPI_File_delete(fname,MPI_INFO_NULL);
       firstcall=0;
   }else{
       amode = MPI_MODE_WRONLY | MPI_MODE_APPEND;
   }
   MPI_File_open(MPI_COMM_WORLD,fname,amode,MPI_INFO_NULL,&fh);

   if (rank==0) {
       double tmp=ng;
       MPI_File_write(fh,&tmp,1,MPI_DOUBLE,MPI_STATUSES_IGNORE);
       MPI_File_write(fh,&time,1,MPI_DOUBLE,MPI_STATUSES_IGNORE);
       MPI_File_write(fh,&H,1,MPI_DOUBLE,MPI_STATUSES_IGNORE);
   }
   MPI_Gather(x,n,MPI_DOUBLE,xbig,n,MPI_DOUBLE,0,MPI_COMM_WORLD);
   if (rank==0) {
```

```
        MPI_File_write(fh,xbig,ng,MPI_DOUBLE,MPI_STATUSES_IGNORE);
    }
    MPI_Gather(y,n,MPI_DOUBLE,xbig,n,MPI_DOUBLE,0,MPI_COMM_WORLD);
    if (rank==0) {
        MPI_File_write(fh,xbig,ng,MPI_DOUBLE,MPI_STATUSES_IGNORE);
    }
    MPI_File_close(&fh);
    return 0;
}
/* eof */
```