Universidad de Investigación de Tecnología Experimental Yachay Tech

School of Mathematics and Computer sciences

Program : Undergraduate ————	Semester : IX
Subject : HPC	Professor : Saravana Prakash T
Date : 13, 15, 16 /05/2019	Time:Monday 11:12:30 hours,
Project session Lab	Wednesday 16-17 hours,
Semester: Jan-May-2019	Thursday, 17:30-20:00 hours,
Student's Surname, Name : Joseph Gonzalez	
Student Reg.No:	

Duration: 10 minutes *Unit V: MPI and OpenMP Final Project* Maximum Marks: 100

Project in HPC : MPI and OpenMPI

1. Your task

Choose any one of the given below problem of 'Physics for Game Programmers, Grant Palmer' and parallelize your problem for both MPI and openMPI methods, your code must include any one of the three following IVP ODE scheme's.

I use the Fourth-Order Runge-Kutta integrator in this work
Programming Drag Effects into the Projectile Trajectory Model

Serial Code

```
#include <stdio.h>
#include <math.h>
#include <malloc.h>
#include <unistd.h>

//*******************************************
//  This structure defines the data required
//  to model a drag projectile
//*******************************************
struct DragProjectile {
   int numEqns;
   double s;
   double q[6];
   double mass;
   double area;
   double density;
   double Cd;
};

```

```

//**********************
//  Function prototypes
//**********************
void projectileRightHandSide(struct DragProjectile *projectile,
                             double *q, double *deltaQ, double ds,
                             double qScale, double *dq);
void projectileRungeKutta4(struct DragProjectile *projectile, double ds);


//**************************************************
//  Main method. It initializes a drag projectile and solves
//  for the golf ball motion using the Runge-Kutta solver
//**************************************************
int main(int argc, char *argv[]) {

  struct DragProjectile golfball;
  double time;
  double x;
  double z;
  double vz;
  double dt = 0.1;

  //  Initialize golfball parameters
  golfball.mass = 0.0459;
  golfball.area = 0.001432;
  golfball.density = 1.225;
  golfball.Cd = 0.25;
  golfball.numEqns = 6;
  golfball.s = 0.0;      //  time = 0.0
  golfball.q[0] = 31.0;  //  vx = 31.0
  golfball.q[1] = 0.0;   //  x  = 0.0
  golfball.q[2] = 0.0;   //  vy = 0.0
  golfball.q[3] = 0.0;   //  y  = 0.0
  golfball.q[4] = 35.0;  //  vz = 35.0
  golfball.q[5] = 0.0;   //  z  = 0.0

  //  Fly the golf ball until z<0
  while ( golfball.q[5] >= 0.0 ) {
    projectileRungeKutta4(&golfball, dt);

    time = golfball.s;
    x = golfball.q[1];
    vz = golfball.q[4];
    z = golfball.q[5];
    printf("time = %lf  x = %lf  z = %lf  vz = %lf\n",
            time, x, z, vz);
    // sleep(2);
  }

  return 0;
}

//*******************************************************
//  This method solves for the projectile motion using a
//  4th-order Runge-Kutta solver
//*******************************************************
```

```c
void projectileRungeKutta4(struct DragProjectile *projectile, double ds) {

  int j;
  int numEqns;
  double s;
  double *q;
  double *dq1;
  double *dq2;
  double *dq3;
  double *dq4;

  //  Define a convenience variable to make the
  //  code more readable
  numEqns = projectile->numEqns;

  //  Allocate memory for the arrays.
  q = (double *)malloc(numEqns*sizeof(double));
  dq1 = (double *)malloc(numEqns*sizeof(double));
  dq2 = (double *)malloc(numEqns*sizeof(double));
  dq3 = (double *)malloc(numEqns*sizeof(double));
  dq4 = (double *)malloc(numEqns*sizeof(double));

  //  Retrieve the current values of the dependent
  //  and independent variables.
  s = projectile->s;
  for(j=0; j<numEqns; ++j) {
    q[j] = projectile->q[j];
  }

  // Compute the four Runge-Kutta steps, The return
  // value of projectileRightHandSide method is an array
  // of delta-q values for each of the four steps.
  projectileRightHandSide(projectile, q, q,   ds, 0.0, dq1);
  projectileRightHandSide(projectile, q, dq1, ds, 0.5, dq2);
  projectileRightHandSide(projectile, q, dq2, ds, 0.5, dq3);
  projectileRightHandSide(projectile, q, dq3, ds, 1.0, dq4);

  //  Update the dependent and independent variable values
  //  at the new dependent variable location and store the
  //  values in the ODE object arrays.
  projectile->s = projectile->s + ds;

  // for(int i=0; i<6; i++){
  //     printf("q-> %f, q1 -> %f, q2-> %f, q3 -> %f, q4-> %f \n",q[i],dq1[i],dq2[i],dq3[i],dq4[i]);
  // }

  for(j=0; j<numEqns; ++j) {
    q[j] = q[j] + (dq1[j] + 2.0*dq2[j] + 2.0*dq3[j] + dq4[j])/6.0;
    projectile->q[j] = q[j];
  }

  //  Free up memory
  free(q);
  free(dq1);
  free(dq2);
  free(dq3);
```

```c
  free(dq4);

  return;
}

//**********************************************************
//  This method loads the right-hand sides for the spring ODEs
//**********************************************************
void projectileRightHandSide(struct DragProjectile *projectile,
                             double *q, double *deltaQ, double ds,
                             double qScale, double *dq) {
  //  q[0] = vx = dxdt
  //  q[1] = x
  //  q[2] = vy = dydt
  //  q[3] = y
  //  q[4] = vz = dzdt
  //  q[5] = z
  double newQ[6]; // intermediate dependent variable values.
  double mass;
  double area;
  double density;
  double Cd;
  double vx;
  double vy;
  double vz;
  double v;
  double Fd;
  double G = -9.81;

  int i;

  mass = projectile->mass;
  area = projectile->area;
  density = projectile->density;
  Cd = projectile->Cd;

  //  Compute the intermediate values of the
  //  dependent variables.
  for(i=0; i<6; ++i) {
    newQ[i] = q[i] + qScale*deltaQ[i];
  }

  //  Declare some convenience variables representing
  //  the intermediate values of velocity.
  vx = newQ[0];
  vy = newQ[2];
  vz = newQ[4];

  //  Compute the velocity magnitude. The 1.0e-8 term
  //  ensures there won't be a divide by zero later on
  //  if all of the velocity components are zero.
  v = sqrt(vx*vx + vy*vy + vz*vz) + 1.0e-8;

  //  Compute the total drag force.
  Fd = 0.5*density*area*Cd*v*v;

  //  Compute right-hand side values.
```

```
  dq[0] = -ds*Fd*vx/(mass*v);
  dq[1] = ds*vx;
  dq[2] = -ds*Fd*vy/(mass*v);
  dq[3] = ds*vy;
  dq[4] = ds*(G - Fd*vz/(mass*v));
  dq[5] = ds*vz;

  return;
}
```

Parallel Code using MPI

```
#include <stdio.h>
#include <math.h>
#include <malloc.h>
#include <mpi.h>

//******************************************
//  This structure defines the data required
//  to model a drag projectile
//******************************************
struct DragProjectile {
   int numEqns;
   double s;
   double q[6];
   double mass;
   double area;
   double density;
   double Cd;
};


//**********************
//  Function prototypes
//**********************
void projectileRightHandSide(struct DragProjectile *projectile,
                             double *q, double *deltaQ, double ds,
                             double qScale, double *dq);
void projectileRungeKutta4(struct DragProjectile *projectile, double ds, int
    world_rank);


//**************************************************
//  Main method. It initializes a drag projectile and solves
//  for the golf ball motion using the Runge-Kutta solver
//**************************************************
int main(int argc, char *argv[]) {

    MPI_Init(NULL,NULL);
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    double z_floor = 0.0;
    struct DragProjectile golfball;
    double dt = 0.1;
    double time;
```

```
    double x;
    double z;
    double vz;

    if(world_rank == 0){
        //  Initialize golfball parameters
        golfball.mass = 0.0459;
        golfball.area = 0.001432;
        golfball.density = 1.225;
        golfball.Cd = 0.25;
        golfball.numEqns = 6;
        golfball.s = 0.0;       //  time = 0.0
        golfball.q[0] = 31.0;   //  vx = 31.0
        golfball.q[1] = 0.0;    //  x  = 0.0
        golfball.q[2] = 0.0;    //  vy = 0.0
        golfball.q[3] = 0.0;    //  y  = 0.0
        golfball.q[4] = 35.0;   //  vz = 35.0
        golfball.q[5] = 0.0;    //  z  = 0.0
    }


    //  Fly the golf ball until z<0
    while ( z_floor >= 0.0 ) {

        projectileRungeKutta4(&golfball, dt, world_rank);

        if(world_rank==0){
            time = golfball.s;
            x = golfball.q[1];
            vz = golfball.q[4];
            z = golfball.q[5];
            printf("time = %lf   x = %lf   z = %lf   vz = %lf\n",
                    time, x, z, vz);
            for(int i=0; i<world_size;i++){
                MPI_Send(&z,1,MPI_DOUBLE,i,0, MPI_COMM_WORLD);
            }
        }

        // MPI_Recv(data,count,datatype,source,tag,communicator,status);
        MPI_Recv(&z_floor,1,MPI_DOUBLE,0,0, MPI_COMM_WORLD,MPI_STATUS_IGNORE);

    }

    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Finalize();
    return 0;
}

//*********************************************************
//  This method solves for the projectile motion using a
//  4th-order Runge-Kutta solver
//*********************************************************
void projectileRungeKutta4(struct DragProjectile *projectile, double ds, int world_rank) {

    double rec_q,rec_q1,rec_q2,rec_q3,rec_q4, new_q;

```

```c
    double *q = NULL;
    double *dq1 = NULL;
    double *dq2 = NULL;
    double *dq3 = NULL;
    double *dq4 = NULL;

    if(world_rank==0){
        int j;
        int numEqns;
        double s;

        //  Define a convenience variable to make the
        //  code more readable
        numEqns = projectile->numEqns;

        //  Allocate memory for the arrays.
        q = (double *)malloc(numEqns*sizeof(double));
        dq1 = (double *)malloc(numEqns*sizeof(double));
        dq2 = (double *)malloc(numEqns*sizeof(double));
        dq3 = (double *)malloc(numEqns*sizeof(double));
        dq4 = (double *)malloc(numEqns*sizeof(double));

        //  Retrieve the current values of the dependent
        //  and independent variables.
        s = projectile->s;
        for(j=0; j<numEqns; ++j) {
            q[j] = projectile->q[j];
        }

        // Compute the four Runge-Kutta steps, The return
        // value of projectileRightHandSide method is an array
        // of delta-q values for each of the four steps.
        projectileRightHandSide(projectile, q, q,   ds, 0.0, dq1);
        projectileRightHandSide(projectile, q, dq1, ds, 0.5, dq2);
        projectileRightHandSide(projectile, q, dq2, ds, 0.5, dq3);
        projectileRightHandSide(projectile, q, dq3, ds, 1.0, dq4);

        //  Update the dependent and independent variable values
        //  at the new dependent variable location and store the
        //  values in the ODE object arrays.
        projectile->s = projectile->s + ds;
        for(int i=0; i<6; i++){
            printf("q-> %f, q1 -> %f, q2-> %f, q3 -> %f, q4-> %f \n",q[i],dq1[i],dq2[i],dq3[i],dq4[i]);
        }
    }

    // MPI_Scatter(send_data,send_count,send_datatype, recv_data,recv_count,recv_datatype,root,communicator)
    MPI_Scatter(q,1,MPI_DOUBLE,&rec_q  ,1,MPI_DOUBLE,0,MPI_COMM_WORLD);
    MPI_Scatter(dq1,1,MPI_DOUBLE,&rec_q1  ,1,MPI_DOUBLE,0,MPI_COMM_WORLD);
    MPI_Scatter(dq2,1,MPI_DOUBLE,&rec_q2  ,1,MPI_DOUBLE,0,MPI_COMM_WORLD);
    MPI_Scatter(dq3,1,MPI_DOUBLE,&rec_q3  ,1,MPI_DOUBLE,0,MPI_COMM_WORLD);
    MPI_Scatter(dq4,1,MPI_DOUBLE,&rec_q4  ,1,MPI_DOUBLE,0,MPI_COMM_WORLD);

    new_q = rec_q + (rec_q1 + 2.0*rec_q2 + 2.0*rec_q3 + rec_q4)/6.0;

```

```
    // MPI_Gather(send_data,send_count,send_datatype,recv_data,recv_count,
    recv_datatype,root,communicator)
    MPI_Gather(&new_q,1,MPI_DOUBLE, projectile->q,1,MPI_DOUBLE,0,MPI_COMM_WORLD);

    if(world_rank==0){
        //  Free up memory
        free(q);
        free(dq1);
        free(dq2);
        free(dq3);
        free(dq4);
    }
    return;
}

//**********************************************************
//  This method loads the right-hand sides for the spring ODEs
//**********************************************************
void projectileRightHandSide(struct DragProjectile *projectile,
                             double *q, double *deltaQ, double ds,
                             double qScale, double *dq) {
    //  q[0] = vx = dxdt
    //  q[1] = x
    //  q[2] = vy = dydt
    //  q[3] = y
    //  q[4] = vz = dzdt
    //  q[5] = z
    double newQ[6]; // intermediate dependent variable values.
    double mass;
    double area;
    double density;
    double Cd;
    double vx;
    double vy;
    double vz;
    double v;
    double Fd;
    double G = -9.81;

    int i;

    mass = projectile->mass;
    area = projectile->area;
    density = projectile->density;
    Cd = projectile->Cd;

    //  Compute the intermediate values of the
    //  dependent variables.
    for(i=0; i<6; ++i) {
        newQ[i] = q[i] + qScale*deltaQ[i];
    }

    //  Declare some convenience variables representing
    //  the intermediate values of velocity.
    vx = newQ[0];
    vy = newQ[2];
    vz = newQ[4];
```

```c

    //  Compute the velocity magnitude. The 1.0e-8 term
    //  ensures there won't be a divide by zero later on
    //  if all of the velocity components are zero.
    v = sqrt(vx*vx + vy*vy + vz*vz) + 1.0e-8;

    //  Compute the total drag force.
    Fd = 0.5*density*area*Cd*v*v;

    //  Compute right-hand side values.
    dq[0] = -ds*Fd*vx/(mass*v);
    dq[1] = ds*vx;
    dq[2] = -ds*Fd*vy/(mass*v);
    dq[3] = ds*vy;
    dq[4] = ds*(G - Fd*vz/(mass*v));
    dq[5] = ds*vz;

    return;
}
```

Parallel Code using OpenMP

```c
#include<stdio.h>
#include<math.h>
#include<malloc.h>
#include<unistd.h>
#include<omp.h>

//*******************************************
//  This structure defines the data required
//  to model a drag projectile
//*******************************************
struct DragProjectile {
   int numEqns;
   double s;
   double q[6];
   double mass;
   double area;
   double density;
   double Cd;
};


//**********************
//  Function prototypes
//**********************
void projectileRightHandSide(struct DragProjectile *projectile,
                             double *q, double *deltaQ, double ds,
                             double qScale, double *dq);
void projectileRungeKutta4(struct DragProjectile *projectile, double ds);


//*************************************************
//  Main method. It initializes a drag projectile and solves
//  for the golf ball motion using the Runge-Kutta solver
//*************************************************
int main(int argc, char *argv[]) {

```

```
    struct DragProjectile golfball;
    double time;
    double x;
    double z;
    double vz;
    double dt = 0.1;

    //  Initialize golfball parameters
    golfball.mass = 0.0459;
    golfball.area = 0.001432;
    golfball.density = 1.225;
    golfball.Cd = 0.25;
    golfball.numEqns = 6;
    golfball.s = 0.0;       //  time = 0.0
    golfball.q[0] = 31.0;   //  vx = 31.0
    golfball.q[1] = 0.0;    //  x  = 0.0
    golfball.q[2] = 0.0;    //  vy = 0.0
    golfball.q[3] = 0.0;    //  y  = 0.0
    golfball.q[4] = 35.0;   //  vz = 35.0
    golfball.q[5] = 0.0;    //  z  = 0.0

    //  Fly the golf ball until z<0
    while ( golfball.q[5] >= 0.0 ) {
        projectileRungeKutta4(&golfball, dt);

        time = golfball.s;
        x = golfball.q[1];
        vz = golfball.q[4];
        z = golfball.q[5];
        printf("time = %lf   x = %lf   z = %lf   vz = %lf\n",
                time, x, z, vz);
        // sleep(2);
    }

    return 0;
}

//**********************************************************
//  This method solves for the projectile motion using a
//  4th-order Runge-Kutta solver
//**********************************************************
void projectileRungeKutta4(struct DragProjectile *projectile, double ds) {

    int j;
    int numEqns;
    double s;
    double *q;
    double *dq1;
    double *dq2;
    double *dq3;
    double *dq4;

    //  Define a convenience variable to make the
    //  code more readable
    numEqns = projectile->numEqns;

    //  Allocate memory for the arrays.
```

```
    q = (double *)malloc(numEqns*sizeof(double));
    dq1 = (double *)malloc(numEqns*sizeof(double));
    dq2 = (double *)malloc(numEqns*sizeof(double));
    dq3 = (double *)malloc(numEqns*sizeof(double));
    dq4 = (double *)malloc(numEqns*sizeof(double));

    //  Retrieve the current values of the dependent
    //  and independent variables.
    s = projectile->s;
    for(j=0; j<numEqns; ++j) {
        q[j] = projectile->q[j];
    }

    // Compute the four Runge-Kutta steps, The return
    // value of projectileRightHandSide method is an array
    // of delta-q values for each of the four steps.
    projectileRightHandSide(projectile, q, q,   ds, 0.0, dq1);
    projectileRightHandSide(projectile, q, dq1, ds, 0.5, dq2);
    projectileRightHandSide(projectile, q, dq2, ds, 0.5, dq3);
    projectileRightHandSide(projectile, q, dq3, ds, 1.0, dq4);

    //  Update the dependent and independent variable values
    //  at the new dependent variable location and store the
    //  values in the ODE object arrays.
    projectile->s = projectile->s + ds;

    // for(int i=0; i<6; i++){
    //     printf("q-> %f, q1 -> %f, q2-> %f, q3 -> %f, q4-> %f \n",q[i],dq1[i],dq2[i],dq3[i],dq4[i]);
    // }


    #pragma parallel for num_threads(numEqns)
    {
        for(j=0; j<numEqns; ++j) {
            q[j] = q[j] + (dq1[j] + 2.0*dq2[j] + 2.0*dq3[j] + dq4[j])/6.0;
            projectile->q[j] = q[j];
        }
    }


    //  Free up memory
    free(q);
    free(dq1);
    free(dq2);
    free(dq3);
    free(dq4);

    return;
}

//**********************************************************
//  This method loads the right-hand sides for the spring ODEs
//**********************************************************
void projectileRightHandSide(struct DragProjectile *projectile,
                             double *q, double *deltaQ, double ds,
                             double qScale, double *dq) {
```

```
    //  q[0] = vx = dxdt
    //  q[1] = x
    //  q[2] = vy = dydt
    //  q[3] = y
    //  q[4] = vz = dzdt
    //  q[5] = z
    double newQ[6]; // intermediate dependent variable values.
    double mass;
    double area;
    double density;
    double Cd;
    double vx;
    double vy;
    double vz;
    double v;
    double Fd;
    double G = -9.81;

    int i;

    mass = projectile->mass;
    area = projectile->area;
    density = projectile->density;
    Cd = projectile->Cd;

    //  Compute the intermediate values of the
    //  dependent variables.
    for(i=0; i<6; ++i) {
        newQ[i] = q[i] + qScale*deltaQ[i];
    }

    //  Declare some convenience variables representing
    //  the intermediate values of velocity.
    vx = newQ[0];
    vy = newQ[2];
    vz = newQ[4];

    //  Compute the velocity magnitude. The 1.0e-8 term
    //  ensures there won't be a divide by zero later on
    //  if all of the velocity components are zero.
    v = sqrt(vx*vx + vy*vy + vz*vz) + 1.0e-8;

    //  Compute the total drag force.
    Fd = 0.5*density*area*Cd*v*v;

    //  Compute right-hand side values.
    dq[0] = -ds*Fd*vx/(mass*v);
    dq[1] = ds*vx;
    dq[2] = -ds*Fd*vy/(mass*v);
    dq[3] = ds*vy;
    dq[4] = ds*(G - Fd*vz/(mass*v));
    dq[5] = ds*vz;

    return;
}
```

2. Table results

	Time Step				
Runge-Kutta	**1**	**0.1**	**0.01**	**0.001**	**0.0001**
Serial	0.001	0.004	0.008	0.035	0.245
MPI	0.234	0.264	0.383	1.447	8.503
OpenMp	0.002	0.001	0.006	0.028	0.231

Cuadro 1: Runge-Kutta method for serial, mpi and openMP with different time steps

This table shows strange results because Serial code seems to be so fast as OpenMP implementation, this could be because this algorithm cannot be strongly paralleled, in other words this algorithm has a serial part very large, another important result is that MPI seems to be the worst alternative for this purpose.