

Program : Undergraduate \_\_\_\_\_  
Subject : HPC  
Date : 13, 15, 16 /05/2019  
Project session Lab  
Semester: Jan-May-2019  
Student's Surname, Name : Joseph Gonzalez  
Student Reg.No:

Semester : IX  
Professor : Saravana Prakash T  
Time:Monday 11:12:30 hours,  
Wednesday 16-17 hours,  
Thursday, 17:30-20:00 hours,

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

```
1 #include <stdio.h>
2 #include <math.h>
3 #include <malloc.h>
4 #include <unistd.h>
5
6 // *****
7 // This structure defines the data required
8 // to model a drag projectile
9 // *****
10 struct DragProjectile {
11     int numEqns;
12     double s;
13     double q[6];
14     double mass;
15     double area;
16     double density;
17     double Cd;
18 };
19
```

```

20
21 //*****
22 //  Function prototypes
23 //*****
24 void projectileRightHandSide(struct DragProjectile *projectile ,
25                             double *q, double *deltaQ, double ds,
26                             double qScale, double *dq);
27 void projectileRungeKutta4(struct DragProjectile *projectile , double ds);
28
29
30 //*****
31 //  Main method. It initializes a drag projectile and solves
32 //  for the golf ball motion using the Runge–Kutta solver
33 //*****
34 int main(int argc, char *argv[]) {
35
36     struct DragProjectile golfball;
37     double time;
38     double x;
39     double z;
40     double vz;
41     double dt = 0.1;
42
43     //  Initialize golfball parameters
44     golfball.mass = 0.0459;
45     golfball.area = 0.001432;
46     golfball.density = 1.225;
47     golfball.Cd = 0.25;
48     golfball.numEqns = 6;
49     golfball.s = 0.0;    //  time = 0.0
50     golfball.q[0] = 31.0; //  vx = 31.0
51     golfball.q[1] = 0.0;  //  x  = 0.0
52     golfball.q[2] = 0.0;  //  vy = 0.0
53     golfball.q[3] = 0.0;  //  y  = 0.0
54     golfball.q[4] = 35.0; //  vz = 35.0
55     golfball.q[5] = 0.0;  //  z  = 0.0
56
57     //  Fly the golf ball until z<0
58     while ( golfball.q[5] >= 0.0 ) {
59         projectileRungeKutta4(&golfball , dt);
60
61         time = golfball.s;
62         x = golfball.q[1];
63         vz = golfball.q[4];
64         z = golfball.q[5];
65         printf("time = %f  x = %f  z = %f  vz = %f\n",
66              time, x, z, vz);
67         //  sleep(2);
68     }
69
70     return 0;
71 }
72
73 //*****
74 //  This method solves for the projectile motion using a
75 //  4th–order Runge–Kutta solver
76 //*****

```

```

77 void projectileRungeKutta4(struct DragProjectile *projectile, double ds) {
78
79     int j;
80     int numEqns;
81     double s;
82     double *q;
83     double *dq1;
84     double *dq2;
85     double *dq3;
86     double *dq4;
87
88     // Define a convenience variable to make the
89     // code more readable
90     numEqns = projectile->numEqns;
91
92     // Allocate memory for the arrays.
93     q = (double *)malloc(numEqns*sizeof(double));
94     dq1 = (double *)malloc(numEqns*sizeof(double));
95     dq2 = (double *)malloc(numEqns*sizeof(double));
96     dq3 = (double *)malloc(numEqns*sizeof(double));
97     dq4 = (double *)malloc(numEqns*sizeof(double));
98
99     // Retrieve the current values of the dependent
100    // and independent variables.
101    s = projectile->s;
102    for(j=0; j<numEqns; ++j) {
103        q[j] = projectile->q[j];
104    }
105
106    // Compute the four Runge-Kutta steps, The return
107    // value of projectileRightHandSide method is an array
108    // of delta-q values for each of the four steps.
109    projectileRightHandSide(projectile, q, q, ds, 0.0, dq1);
110    projectileRightHandSide(projectile, q, dq1, ds, 0.5, dq2);
111    projectileRightHandSide(projectile, q, dq2, ds, 0.5, dq3);
112    projectileRightHandSide(projectile, q, dq3, ds, 1.0, dq4);
113
114    // Update the dependent and independent variable values
115    // at the new dependent variable location and store the
116    // values in the ODE object arrays.
117    projectile->s = projectile->s + ds;
118
119    // for(int i=0; i<6; i++){
120    //     printf("q-> %f, q1-> %f, q2-> %f, q3-> %f, q4-> %f \n",q[i],dq1[i],dq2[i]
121    //     ],dq3[i],dq4[i]);
122    // }
123
124    for(j=0; j<numEqns; ++j) {
125        q[j] = q[j] + (dq1[j] + 2.0*dq2[j] + 2.0*dq3[j] + dq4[j])/6.0;
126        projectile->q[j] = q[j];
127    }
128
129    // Free up memory
130    free(q);
131    free(dq1);
132    free(dq2);
133    free(dq3);

```

```

133 free(dq4);
134
135 return;
136 }
137
138 //*****
139 // This method loads the right-hand sides for the spring ODEs
140 //*****
141 void projectileRightHandSide(struct DragProjectile *projectile,
142                             double *q, double *deltaQ, double ds,
143                             double qScale, double *dq) {
144     // q[0] = vx = dxdt
145     // q[1] = x
146     // q[2] = vy = dydt
147     // q[3] = y
148     // q[4] = vz = dzdt
149     // q[5] = z
150     double newQ[6]; // intermediate dependent variable values.
151     double mass;
152     double area;
153     double density;
154     double Cd;
155     double vx;
156     double vy;
157     double vz;
158     double v;
159     double Fd;
160     double G = -9.81;
161
162     int i;
163
164     mass = projectile->mass;
165     area = projectile->area;
166     density = projectile->density;
167     Cd = projectile->Cd;
168
169     // Compute the intermediate values of the
170     // dependent variables.
171     for(i=0; i<6; ++i) {
172         newQ[i] = q[i] + qScale*deltaQ[i];
173     }
174
175     // Declare some convenience variables representing
176     // the intermediate values of velocity.
177     vx = newQ[0];
178     vy = newQ[2];
179     vz = newQ[4];
180
181     // Compute the velocity magnitude. The 1.0e-8 term
182     // ensures there won't be a divide by zero later on
183     // if all of the velocity components are zero.
184     v = sqrt(vx*vx + vy*vy + vz*vz) + 1.0e-8;
185
186     // Compute the total drag force.
187     Fd = 0.5*density*area*Cd*v*v;
188
189     // Compute right-hand side values.

```

```

190 dq[0] = -ds*Fd*vx/(mass*v);
191 dq[1] = ds*vx;
192 dq[2] = -ds*Fd*vy/(mass*v);
193 dq[3] = ds*vy;
194 dq[4] = ds*(G - Fd*vz/(mass*v));
195 dq[5] = ds*vz;
196
197 return;
198 }

```

### Parallel Code using MPI

```

1 #include <stdio.h>
2 #include <math.h>
3 #include <malloc.h>
4 #include <mpi.h>
5
6 //*****
7 // This structure defines the data required
8 // to model a drag projectile
9 //*****
10 struct DragProjectile {
11     int numEqns;
12     double s;
13     double q[6];
14     double mass;
15     double area;
16     double density;
17     double Cd;
18 };
19
20
21 //*****
22 // Function prototypes
23 //*****
24 void projectileRightHandSide(struct DragProjectile *projectile,
25                             double *q, double *deltaQ, double ds,
26                             double qScale, double *dq);
27 void projectileRungeKutta4(struct DragProjectile *projectile, double ds, int
28                             world_rank);
29
30 //*****
31 // Main method. It initializes a drag projectile and solves
32 // for the golf ball motion using the Runge-Kutta solver
33 //*****
34 int main(int argc, char *argv[]) {
35
36     MPI_Init(NULL, NULL);
37     int world_rank;
38     MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
39     int world_size;
40     MPI_Comm_size(MPI_COMM_WORLD, &world_size);
41
42     double z_floor = 0.0;
43     struct DragProjectile golfball;
44     double dt = 0.1;
45     double time;

```

```

46  double x;
47  double z;
48  double vz;
49
50  if(world_rank == 0){
51      // Initialize golfball parameters
52      golfball.mass = 0.0459;
53      golfball.area = 0.001432;
54      golfball.density = 1.225;
55      golfball.Cd = 0.25;
56      golfball.numEqns = 6;
57      golfball.s = 0.0;      // time = 0.0
58      golfball.q[0] = 31.0;  // vx = 31.0
59      golfball.q[1] = 0.0;   // x = 0.0
60      golfball.q[2] = 0.0;   // vy = 0.0
61      golfball.q[3] = 0.0;   // y = 0.0
62      golfball.q[4] = 35.0;  // vz = 35.0
63      golfball.q[5] = 0.0;   // z = 0.0
64  }
65
66
67  // Fly the golf ball until z<0
68  while ( z_floor >= 0.0 ) {
69
70      projectileRungeKutta4(&golfball , dt , world_rank);
71
72      if(world_rank==0){
73          time = golfball.s;
74          x = golfball.q[1];
75          vz = golfball.q[4];
76          z = golfball.q[5];
77          printf("time = %f  x = %f  z = %f  vz = %f\n" ,
78              time , x , z , vz);
79          for(int i=0; i<world_size;i++){
80              MPI_Send(&z,1,MPLDOUBLE,i,0, MPLCOMM_WORLD);
81          }
82      }
83
84      // MPI_Recv(data,count,datatype,source,tag,communicator,status);
85      MPI_Recv(&z_floor,1,MPLDOUBLE,0,0, MPLCOMM_WORLD,MPLSTATUS_IGNORE);
86
87  }
88
89  MPI_Barrier(MPLCOMM_WORLD);
90  MPI_Finalize();
91  return 0;
92 }
93
94 //*****
95 // This method solves for the projectile motion using a
96 // 4th-order Runge-Kutta solver
97 //*****
98 void projectileRungeKutta4(struct DragProjectile *projectile, double ds, int
    world_rank) {
99
100     double rec_q ,rec_q1 ,rec_q2 ,rec_q3 ,rec_q4 , new_q;
101

```

```

102 double *q = NULL;
103 double *dq1 = NULL;
104 double *dq2 = NULL;
105 double *dq3 = NULL;
106 double *dq4 = NULL;
107
108 if(world_rank==0){
109     int j;
110     int numEqns;
111     double s;
112
113     // Define a convenience variable to make the
114     // code more readable
115     numEqns = projectile->numEqns;
116
117     // Allocate memory for the arrays.
118     q = (double *)malloc(numEqns*sizeof(double));
119     dq1 = (double *)malloc(numEqns*sizeof(double));
120     dq2 = (double *)malloc(numEqns*sizeof(double));
121     dq3 = (double *)malloc(numEqns*sizeof(double));
122     dq4 = (double *)malloc(numEqns*sizeof(double));
123
124     // Retrieve the current values of the dependent
125     // and independent variables.
126     s = projectile->s;
127     for(j=0; j<numEqns; ++j) {
128         q[j] = projectile->q[j];
129     }
130
131     // Compute the four Runge-Kutta steps, The return
132     // value of projectileRightHandSide method is an array
133     // of delta-q values for each of the four steps.
134     projectileRightHandSide(projectile, q, q, ds, 0.0, dq1);
135     projectileRightHandSide(projectile, q, dq1, ds, 0.5, dq2);
136     projectileRightHandSide(projectile, q, dq2, ds, 0.5, dq3);
137     projectileRightHandSide(projectile, q, dq3, ds, 1.0, dq4);
138
139     // Update the dependent and independent variable values
140     // at the new dependent variable location and store the
141     // values in the ODE object arrays.
142     projectile->s = projectile->s + ds;
143     for(int i=0; i<6; i++){
144         printf("q-> %f, q1-> %f, q2-> %f, q3-> %f, q4-> %f \n",q[i],dq1[i],dq2
145 [i],dq3[i],dq4[i]);
146     }
147
148     // MPI_Scatter(send_data,send_count,send_datatype,recv_data,recv_count,
149     // recv_datatype,root,communicator)
150     MPI_Scatter(q,1,MPLDOUBLE,&rec_q,1,MPLDOUBLE,0,MPLCOMM_WORLD);
151     MPI_Scatter(dq1,1,MPLDOUBLE,&rec_q1,1,MPLDOUBLE,0,MPLCOMM_WORLD);
152     MPI_Scatter(dq2,1,MPLDOUBLE,&rec_q2,1,MPLDOUBLE,0,MPLCOMM_WORLD);
153     MPI_Scatter(dq3,1,MPLDOUBLE,&rec_q3,1,MPLDOUBLE,0,MPLCOMM_WORLD);
154     MPI_Scatter(dq4,1,MPLDOUBLE,&rec_q4,1,MPLDOUBLE,0,MPLCOMM_WORLD);
155
156     new_q = rec_q + (rec_q1 + 2.0*rec_q2 + 2.0*rec_q3 + rec_q4)/6.0;

```

```

157 // MPI_Gather(send_data , send_count , send_datatype , recv_data , recv_count ,
158 //            recv_datatype , root , communicator)
159 MPI_Gather(&new_q , 1 , MPLDOUBLE , projectile -> q , 1 , MPLDOUBLE , 0 , MPLCOMM_WORLD) ;
160
161 if ( world_rank == 0 ) {
162     // Free up memory
163     free ( q ) ;
164     free ( dq1 ) ;
165     free ( dq2 ) ;
166     free ( dq3 ) ;
167     free ( dq4 ) ;
168 }
169 return ;
170 }
171
172 // *****
173 // This method loads the right-hand sides for the spring ODEs
174 // *****
175 void projectileRightHandSide ( struct DragProjectile * projectile ,
176                               double * q , double * deltaQ , double ds ,
177                               double qScale , double * dq ) {
178
179     // q[0] = vx = dxdt
180     // q[1] = x
181     // q[2] = vy = dydt
182     // q[3] = y
183     // q[4] = vz = dzdt
184     // q[5] = z
185     double newQ[6] ; // intermediate dependent variable values.
186     double mass ;
187     double area ;
188     double density ;
189     double Cd ;
190     double vx ;
191     double vy ;
192     double vz ;
193     double v ;
194     double Fd ;
195     double G = -9.81 ;
196
197     int i ;
198
199     mass = projectile -> mass ;
200     area = projectile -> area ;
201     density = projectile -> density ;
202     Cd = projectile -> Cd ;
203
204     // Compute the intermediate values of the
205     // dependent variables.
206     for ( i = 0 ; i < 6 ; ++i ) {
207         newQ[i] = q[i] + qScale * deltaQ[i] ;
208     }
209
210     // Declare some convenience variables representing
211     // the intermediate values of velocity.
212     vx = newQ[0] ;
213     vy = newQ[2] ;
214     vz = newQ[4] ;

```



```

213
214 // Compute the velocity magnitude. The 1.0e-8 term
215 // ensures there won't be a divide by zero later on
216 // if all of the velocity components are zero.
217 v = sqrt(vx*vx + vy*vy + vz*vz) + 1.0e-8;
218
219 // Compute the total drag force.
220 Fd = 0.5*density*area*Cd*v*v;
221
222 // Compute right-hand side values.
223 dq[0] = -ds*Fd*vx/(mass*v);
224 dq[1] = ds*vx;
225 dq[2] = -ds*Fd*vy/(mass*v);
226 dq[3] = ds*vy;
227 dq[4] = ds*(G - Fd*vz/(mass*v));
228 dq[5] = ds*vz;
229
230 return;
231 }

```

### Parallel Code using OpenMP

```

1 #include <stdio.h>
2 #include <math.h>
3 #include <malloc.h>
4 #include <unistd.h>
5 #include <omp.h>
6
7 //*****
8 // This structure defines the data required
9 // to model a drag projectile
10 //*****
11 struct DragProjectile {
12     int numEqns;
13     double s;
14     double q[6];
15     double mass;
16     double area;
17     double density;
18     double Cd;
19 };
20
21
22 //*****
23 // Function prototypes
24 //*****
25 void projectileRightHandSide(struct DragProjectile *projectile,
26                             double *q, double *deltaQ, double ds,
27                             double qScale, double *dq);
28 void projectileRungeKutta4(struct DragProjectile *projectile, double ds);
29
30
31 //*****
32 // Main method. It initializes a drag projectile and solves
33 // for the golf ball motion using the Runge-Kutta solver
34 //*****
35 int main(int argc, char *argv[]) {
36

```

```

37 struct DragProjectile golfball;
38 double time;
39 double x;
40 double z;
41 double vz;
42 double dt = 0.1;
43
44 // Initialize golfball parameters
45 golfball.mass = 0.0459;
46 golfball.area = 0.001432;
47 golfball.density = 1.225;
48 golfball.Cd = 0.25;
49 golfball.numEqns = 6;
50 golfball.s = 0.0; // time = 0.0
51 golfball.q[0] = 31.0; // vx = 31.0
52 golfball.q[1] = 0.0; // x = 0.0
53 golfball.q[2] = 0.0; // vy = 0.0
54 golfball.q[3] = 0.0; // y = 0.0
55 golfball.q[4] = 35.0; // vz = 35.0
56 golfball.q[5] = 0.0; // z = 0.0
57
58 // Fly the golf ball until z<0
59 while ( golfball.q[5] >= 0.0 ) {
60     projectileRungeKutta4(&golfball, dt);
61
62     time = golfball.s;
63     x = golfball.q[1];
64     vz = golfball.q[4];
65     z = golfball.q[5];
66     printf("time = %f x = %f z = %f vz = %f\n",
67           time, x, z, vz);
68     // sleep(2);
69 }
70
71 return 0;
72 }
73
74 //*****
75 // This method solves for the projectile motion using a
76 // 4th-order Runge-Kutta solver
77 //*****
78 void projectileRungeKutta4(struct DragProjectile *projectile, double ds) {
79
80     int j;
81     int numEqns;
82     double s;
83     double *q;
84     double *dq1;
85     double *dq2;
86     double *dq3;
87     double *dq4;
88
89     // Define a convenience variable to make the
90     // code more readable
91     numEqns = projectile->numEqns;
92
93     // Allocate memory for the arrays.

```

```

94  q = (double *) malloc(numEqns*sizeof(double));
95  dq1 = (double *) malloc(numEqns*sizeof(double));
96  dq2 = (double *) malloc(numEqns*sizeof(double));
97  dq3 = (double *) malloc(numEqns*sizeof(double));
98  dq4 = (double *) malloc(numEqns*sizeof(double));
99
100 // Retrieve the current values of the dependent
101 // and independent variables.
102 s = projectile->s;
103 for(j=0; j<numEqns; ++j) {
104     q[j] = projectile->q[j];
105 }
106
107 // Compute the four Runge-Kutta steps, The return
108 // value of projectileRightHandSide method is an array
109 // of delta-q values for each of the four steps.
110 projectileRightHandSide(projectile, q, q, ds, 0.0, dq1);
111 projectileRightHandSide(projectile, q, dq1, ds, 0.5, dq2);
112 projectileRightHandSide(projectile, q, dq2, ds, 0.5, dq3);
113 projectileRightHandSide(projectile, q, dq3, ds, 1.0, dq4);
114
115 // Update the dependent and independent variable values
116 // at the new dependent variable location and store the
117 // values in the ODE object arrays.
118 projectile->s = projectile->s + ds;
119
120 // for(int i=0; i<6; i++){
121 //     printf("q-> %f, q1 -> %f, q2-> %f, q3 -> %f, q4-> %f \n",q[i],dq1[i],dq2[
122 // i],dq3[i],dq4[i]);
123 // }
124
125 #pragma parallel for num_threads(numEqns)
126 {
127     for(j=0; j<numEqns; ++j) {
128         q[j] = q[j] + (dq1[j] + 2.0*dq2[j] + 2.0*dq3[j] + dq4[j])/6.0;
129         projectile->q[j] = q[j];
130     }
131 }
132
133
134 // Free up memory
135 free(q);
136 free(dq1);
137 free(dq2);
138 free(dq3);
139 free(dq4);
140
141 return;
142 }
143
144 //*****
145 // This method loads the right-hand sides for the spring ODEs
146 //*****
147 void projectileRightHandSide(struct DragProjectile *projectile,
148                             double *q, double *deltaQ, double ds,
149                             double qScale, double *dq) {

```

```

150 // q[0] = vx = dxdt
151 // q[1] = x
152 // q[2] = vy = dydt
153 // q[3] = y
154 // q[4] = vz = dzdt
155 // q[5] = z
156 double newQ[6]; // intermediate dependent variable values.
157 double mass;
158 double area;
159 double density;
160 double Cd;
161 double vx;
162 double vy;
163 double vz;
164 double v;
165 double Fd;
166 double G = -9.81;
167
168 int i;
169
170 mass = projectile->mass;
171 area = projectile->area;
172 density = projectile->density;
173 Cd = projectile->Cd;
174
175 // Compute the intermediate values of the
176 // dependent variables.
177 for(i=0; i<6; ++i) {
178     newQ[i] = q[i] + qScale*deltaQ[i];
179 }
180
181 // Declare some convenience variables representing
182 // the intermediate values of velocity.
183 vx = newQ[0];
184 vy = newQ[2];
185 vz = newQ[4];
186
187 // Compute the velocity magnitude. The 1.0e-8 term
188 // ensures there won't be a divide by zero later on
189 // if all of the velocity components are zero.
190 v = sqrt(vx*vx + vy*vy + vz*vz) + 1.0e-8;
191
192 // Compute the total drag force.
193 Fd = 0.5*density*area*Cd*v*v;
194
195 // Compute right-hand side values.
196 dq[0] = -ds*Fd*vx/(mass*v);
197 dq[1] = ds*vx;
198 dq[2] = -ds*Fd*vy/(mass*v);
199 dq[3] = ds*vy;
200 dq[4] = ds*(G - Fd*vz/(mass*v));
201 dq[5] = ds*vz;
202
203 return;
204 }

```

## 2. Table results

<b>Runge-Kutta</b>	<b>Time Step</b>				
	<b>1</b>	<b>0.1</b>	<b>0.01</b>	<b>0.001</b>	<b>0.0001</b>
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.