Monte carlo y las integrales dobles

Oscar Andrés Rosas Hernández *Facultad de Ciencias, UNAM, CDM*

Para esta tarea teniamos que encontrar la masa asi como el centro de masas de dos laminas.

I. 2 Puntos

Lo primero que teniamos que hacer era programar un codigo en c que fuera capaz de resolver nuestros problemas, a continuacion muestro mis ideas, lo primero era hacerlo sin usar computacion en paralelo:

```c
#include <math.h>
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

typedef long double f64;
const f64 MAX = RAND_MAX;

typedef struct integral {
  f64 a, b, c, d, N;
} integral;

typedef struct data {
  f64 mass, xm, ym;
  f64 (*mass_fn)(f64, f64);
  f64 (*xn_fn)(f64, f64);
  f64 (*yn_fn)(f64, f64);
} data;

f64 f1(f64 x, f64 y) { return x + 2 * y; }
f64 xf1(f64 x, f64 y) { return x * f1(x, y); }
f64 yf1(f64 x, f64 y) { return y * f1(x, y); }

f64 f2(f64 x, f64 y) { return sin(sqrt(x * x + y * y)); }
f64 xf2(f64 x, f64 y) { return x * f2(x, y); }
f64 yf2(f64 x, f64 y) { return y * f2(x, y); }

f64 randf(f64 start, f64 end) {
  const f64 difference = end - start;
  const f64 range = difference * ((f64)rand() / MAX);
  return range + start;
}

f64 double_integral(const integral data, f64 f(f64, f64)) {
  f64 total = 0.0;
  for (int i = 0; i < data.N; ++i) total += f(randf(data.a, data.b), randf(data.c, data.d));

  return ((data.b - data.a) * (data.d - data.c)) / data.N * total;
}

void to_calculations(integral i, const data d) {
  f64 const times[4] = {10, 100, 1000, 10000};

  for (int j = 0; j < 4; ++j) {
    i.N = times[j];
    const f64 mass = double_integral(i, d.mass_fn);
    const f64 xm = double_integral(i, d.xn_fn) / mass;
    const f64 ym = double_integral(i, d.yn_fn) / mass;

    printf("%.0Lf times\n", i.N);
    printf("mass = %Lf\n", mass);
    printf("(xm, yn) = (%Lf, %Lf)\n", xm, ym);

    printf("mass error = %Lf\n", fabsl(d.mass - mass) / fabsl(d.mass));

    f64 base_xm = fabsl(d.xm), base_ym = fabsl(d.ym);
    printf("xm error = %Lf\n", fabsl(d.xm - xm) / (base_xm ? base_xm : 1.0));
    printf("ym error = %Lf\n\n", fabsl(d.ym - ym) / (base_ym ? base_ym : 1.0));
```

```
  }
}

int main() {
  srand(time(0));
  clock_t start = clock();


  // First function
  integral i1 = {.a = 0, .b = 1, .c = 0, .d = 1, .N = 100};
  data d1 = {.mass = 1.5, .xm = 0.5556, .ym = 0.611, .mass_fn = f1, .xn_fn = xf1, .yn_fn = yf1};
  to_calculations(i1, d1);

  // Second function
  integral i2 = {.a = -1, .b = 1, .c = -1, .d = 1, .N = 100};
  data d2 = {.mass = 2.6635, .xm = 0.0, .ym = 0.0, .mass_fn = f2, .xn_fn = xf2, .yn_fn = yf2};

  to_calculations(i2, d2);

  clock_t end = clock();

  f64 clocks = end - start;
  f64 time_taken = clocks / CLOCKS_PER_SEC;
  printf("Time taken: %Lf", time_taken);

  return 0;
}
```

Ahora a añadir la parte paralela:

```
#include <math.h>
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

typedef long double f64;
const f64 MAX = RAND_MAX;
int id, nproc;

typedef struct integral {
  f64 a, b, c, d, N;
} integral;

typedef struct data {
  f64 mass, xm, ym;
  f64 (*mass_fn)(f64, f64);
  f64 (*xn_fn)(f64, f64);
  f64 (*yn_fn)(f64, f64);
} data;

f64 f1(f64 x, f64 y) { return x + 2 * y; }
f64 xf1(f64 x, f64 y) { return x * f1(x, y); }
f64 yf1(f64 x, f64 y) { return y * f1(x, y); }

f64 f2(f64 x, f64 y) { return sin(sqrt(x * x + y * y)); }
f64 xf2(f64 x, f64 y) { return x * f2(x, y); }
f64 yf2(f64 x, f64 y) { return y * f2(x, y); }

f64 randf(f64 start, f64 end) {
  const f64 difference = end - start;
  const f64 range = difference * ((f64)rand() / MAX);
  return range + start;
}

f64 double_integral(const integral data, f64 f(f64, f64)) {
  const int work = data.N / (nproc - 1);
  f64 total = 0.0;

  if (id != 0) {
    f64 partial = 0.0;
    for (int i = 0; i < work; ++i) {
      partial += f(randf(data.a, data.b), randf(data.c, data.d));
    }
    MPI_Send(&partial, 1, MPI_LONG_DOUBLE, 0, 1, MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);
    return 3.0;
```

```
  } else {
    MPI_Status status;
    f64 partial = 0.0;
    for (int i = 1; i < nproc; ++i) {
      MPI_Recv(&partial, 1, MPI_LONG_DOUBLE, MPI_ANY_SOURCE, 1, MPI_COMM_WORLD, &status);
      total += partial;
    }
    MPI_Barrier(MPI_COMM_WORLD);
  }

  return ((data.b - data.a) * (data.d - data.c)) / data.N * total;
}

void to_calculations(integral i, const data d) {
  f64 const times[4] = {10, 100, 1000, 10000};

  for (int j = 0; j < 4; ++j) {
    i.N = times[j];
    const f64 mass = double_integral(i, d.mass_fn);
    const f64 xm = double_integral(i, d.xn_fn) / mass;
    const f64 ym = double_integral(i, d.yn_fn) / mass;

    if (id == 0) {
      printf("%.0Lf times\n", i.N);
      printf("mass = %Lf\n", mass);
      printf("(xm, yn) = (%Lf, %Lf)\n", xm, ym);

      printf("mass error = %Lf\n", fabsl(d.mass - mass) / fabsl(d.mass));

      f64 base_xm = fabsl(d.xm), base_ym = fabsl(d.ym);
      printf("xm error = %Lf\n", fabsl(d.xm - xm) / (base_xm ? base_xm : 1.0));
      printf("ym error = %Lf\n\n", fabsl(d.ym - ym) / (base_ym ? base_ym : 1.0));
    }
  }
}

int main() {
  srand(time(0));
  clock_t start = clock();
  MPI_Init(NULL, NULL);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);

  // First function
  integral i1 = {.a = 0, .b = 1, .c = 0, .d = 1, .N = 100};
  data d1 = {.mass = 1.5, .xm = 0.5556, .ym = 0.611, .mass_fn = f1, .xn_fn = xf1, .yn_fn = yf1};
  to_calculations(i1, d1);

  // Second function
  integral i2 = {.a = -1, .b = 1, .c = -1, .d = 1, .N = 100};
  data d2 = {.mass = 2.6635, .xm = 0.0, .ym = 0.0, .mass_fn = f2, .xn_fn = xf2, .yn_fn = yf2};

  to_calculations(i2, d2);

  if (id == 0) {
    clock_t end = clock();

    f64 clocks = end - start;
    f64 time_taken = clocks / CLOCKS_PER_SEC;
    printf("Time taken: %Lf", time_taken);
  }

  MPI_Finalize();

  return 0;
}
```

La idea es sencilla, hacer una funcion para calcular numero aleatorios en un rango, hacer una funcion para aproximar la integral usando la formula y finalmente mostrar resultados.

Use MPI para dividir segun el ID, siendo este el encargado de recolectar los resultados nada mas y use barrier para esperar a todos los resultados.

Los cuales se pueden correr usando estos comandos:

```
mpicc codeparallel.c -lm -O3 -o code && mpirun -np NUM ./code >b.output
```

II. 4 PUNTOS: RESULTADOS

Ahora podremos ver los resultados de ambos lados a continuacion:

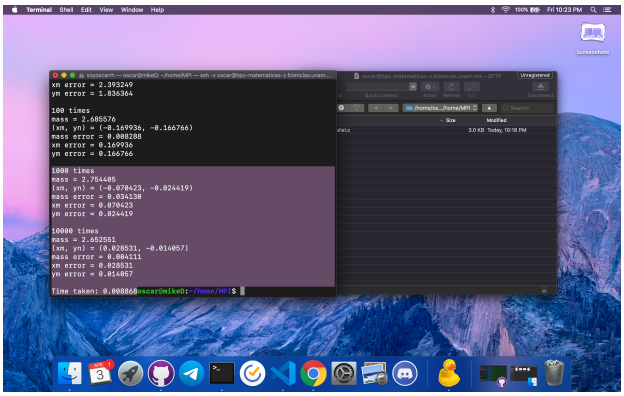
Primero del servidor:

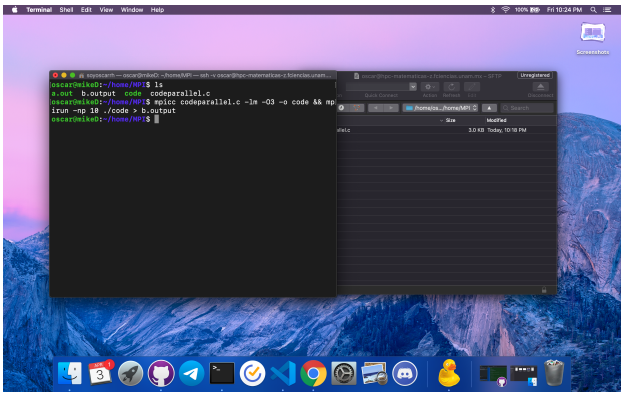
```
10 times
mass = 1.474930
(xm, yn) = (0.807326, 0.819863)
mass error = 0.016713
xm error = 0.453071
ym error = 0.341838

100 times
mass = 1.632575
(xm, yn) = (0.601853, 0.608797)
mass error = 0.088384
xm error = 0.083248
ym error = 0.003605

1000 times
mass = 1.473612
(xm, yn) = (0.581663, 0.606817)
mass error = 0.017592
xm error = 0.046910
ym error = 0.006847

10000 times
mass = 1.498105
(xm, yn) = (0.560489, 0.602842)
mass error = 0.001264
xm error = 0.008800
ym error = 0.013351

10 times
mass = 2.425109
(xm, yn) = (0.209750, 0.131589)
mass error = 0.089503
xm error = 0.209750
ym error = 0.131589

100 times
mass = 2.693056
(xm, yn) = (0.034351, 0.035669)
mass error = 0.011097
xm error = 0.034351
ym error = 0.035669

1000 times
mass = 2.654825
(xm, yn) = (0.032611, 0.001421)
mass error = 0.003257
xm error = 0.032611
ym error = 0.001421

10000 times
mass = 2.666633
(xm, yn) = (0.018292, 0.003600)
mass error = 0.001176
xm error = 0.018292
ym error = 0.003600

Time taken: 0.013798
```

Luego el mio:

```
10 times
mass = 1.462352
(xm, yn) = (0.480025, 0.500611)
mass error = 0.025099
xm error = 0.136025
ym error = 0.180669

100 times
mass = 1.564907
(xm, yn) = (0.631424, 0.506614)
mass error = 0.043272
xm error = 0.136472
ym error = 0.170845

1000 times
mass = 1.484954
(xm, yn) = (0.508084, 0.560789)
mass error = 0.010031
xm error = 0.085523
ym error = 0.082178

10000 times
mass = 1.506648
(xm, yn) = (0.549873, 0.602674)
mass error = 0.004432
xm error = 0.010308
ym error = 0.013627

10 times
mass = 2.665837
(xm, yn) = (0.108239, 0.450652)
mass error = 0.000877
xm error = 0.108239
ym error = 0.450652

100 times
mass = 2.705578
(xm, yn) = (0.123492, -0.039221)
mass error = 0.015798
xm error = 0.123492
ym error = 0.039221

1000 times
mass = 2.625188
(xm, yn) = (0.019555, 0.021004)
mass error = 0.014384
xm error = 0.019555
ym error = 0.021004

10000 times
mass = 2.672327
(xm, yn) = (0.001202, 0.018976)
mass error = 0.003314
xm error = 0.001202
ym error = 0.018976

Time taken: 0.079008
```

Aqui podemos ver varios ejemplos de prueba que hice:





III. 2 PUNTOS: TIEMPO

Ahora resulta que mi programa ya no da el tiempo:

- Con mi computadora: 0.079008
- Con el servidor: 0.013798

Por lo tanto el servidor es unas 7 veces mas rapido a igualdad de procesos (4).

IV. 2 PUNTOS: ERRORES

Mi programa ya calcula los errores pero si lo quieren en una tabla aqui esta:

IV-A. Mi compu

IV-B. Primera funcion de densidad

Masa

N	Aproximacion	Valor real	Error relativo
10	1.462352	1.500000	0.025099
100	1.564907	1.500000	0.043272
1000	1.484954	1.500000	0.010031
10000	1.506648	1.500000	0.004432

x_m

N	Aproximacion	Valor real	Error relativo
10	0.480025	0.5556	0.136025
100	0.631424	0.5556	0.136472
1000	0.508084	0.5556	0.085523
10000	0.549873	0.5556	0.010308

y_m

N	Aproximacion	Valor real	Error relativo
10	0.500611	0.6110	0.180669
100	0.506614	0.6110	0.170845
1000	0.560789	0.6110	0.082178
10000	0.602674	0.6110	0.013627

IV-C. Segunda funcion de densidad

Masa

N	Aproximacion	Valor real	Error relativo
10	2.665837	2.6635	0.000877
100	2.705578	2.6635	0.015798
1000	2.625188	2.6635	0.014384
10000	2.672327	2.6635	0.003314

x_m

N	Aproximacion	Valor real	Error relativo
10	0.108239	0.0000	0.108239
100	0.123492	0.0000	0.123492
1000	0.019555	0.0000	0.019555
10000	0.001202	0.0000	0.001202

y_m

N	Aproximacion	Valor real	Error relativo
10	0.450652	0.0000	0.450652
100	-0.039221	0.0000	0.039221
1000	0.021004	0.0000	0.021004
10000	0.018976	0.0000	0.018976

IV-D. El servidor

Podria escribirlo pero ya esta el archivo de salida, y es bastante tardado hacerlo hacer la tabla en LaTeX