Proyecto final Métodos Numéricos y Optimización

Monte Carlo integración en MPI

30/05/2017

Abboud Camilo(167368), Leblanc Marin(168738)

El objetivo del siguiente trabajo es la implementación de una integración Monte Carlo haciendo uso de MPI para su paralización. Nos enfrentaremos a decisiones de diseño de programación que tendrán un impacto en la precisión y velocidad de la ejecución de nuestro programa.

I) Introducción

A principios de los años ochenta, se pensaba ampliamente que los métodos Monte Carlo no eran adecuados para problemas SIMD y de paralización en general, puesto que lo que se hacía era paralelizar sobre el loop (más) exterior, lo cual hacía que cada partícula se ejecutara de principio a fin, con todas sus ramificaciones y complicaciones, incurriendo en desbalances en la carga de trabajo por hilo.

Ahora ha quedado claro que los métodos Monte Carlo son ideales para entornos paralelos. Se suelen vectorizar los bucles internos dividiendo la muestra k*N en k trozos. Se suelen dar a cada proceso un número de simulación fijo. Las simulaciones locales se pueden distribuir fácilmente en tareas independientes por procesador, lo cual hace a la integración MC un ejemplo clásico de la programación en paralelo. A nuestro parecer, la primera característica, aprovechar de la independencia de los procesos, es la que genera el resto de las ventajas de la programación en paralelo de la integración Monte Carlo.

En la primera parte de nuestro proyecto, presentamos rápidamente la integración de Monte Carlo, la simulación de números aleatorias en la máquina y proponemos una paralización eficiente en una arquitectura distribuida con el objetivo de aumentar el número de simulaciones y la independencia en la muestra final. En la segunda parte explicamos cómo implementamos la integración MC en MPI respectando los conceptos presentados en la primera parte.

II) Teoría:

1) De la simulación de números aleatorios hasta la integración

Nuestro objetivo es calcular una integración multidimensional $I = \iiint g(x,y,z)dxdydz$ sobre $E \subset \mathbb{R}^n$ por el intermedio de simulaciones aleatorias. Primero tenemos que crear una densidad dentro el integral I:

Descomponer g en 2 funciones: $g(x,y,z) = h(x,y,z)l(x,y,z)$

Elegir la función l tal que:

- $l(x,y,z) \geq 0 \ \ \forall (x,y,z) \epsilon E$ y
- l sea continua sobre E
- Sea fácil de calcular su integral sobre E.

Calcular el integral $V = \iiint l(x,y,z)\, dxdydz$

Notamos $p(x,y,z) = \frac{l(x,y,z)}{V}$

p representa una densidad de probabilidad sobre E

$$p: E \to [0,1]$$

$$x,y,z \to p(x,y,z)$$

Tenemos finalmente:

$$I = V \iiint h(x,y,z)p(x,y,z)\, dxdydz$$

Ahora suponemos (x1,y1,z1) ,(x2,y2,z2),….,(xn,yn,zn) representan una muestra independiente de la variable aleatoria siguiendo la densidad p sobre E.

Tenemos 2 teoremas fundamentales:

<u>Ley de los grandes números:</u>

$\lim_{n\to+\infty} \frac{1}{n} \sum_{i=1}^{n} h(x_{i,} y_i, z_i) = E(h(X))$ con X~Uniform(E)

<u>Teorema de transferencia :</u>

$E(g(X)) = \iiint h(x,y,z)p(x,y,z)d(x,y,z)$

En consecuencia con $I = VE(h(X))$ podemos aproximar I con una n grande de simulaciones de (xi,yi,zi) :

$$\frac{V}{n} \sum_{i=1}^{n} h(x_{i,} y_i, z_i) \approx \iiint g(x,y,z)dxdydz$$

Para n más grande mejora nuestra estimación.

Sin embargo necesitamos tener una simulación de un ley de probabilidad de densidad p sobre $E \subset \mathbb{R}^n$.Depende como creemos la función l, podemos separar en 2 casos Monte Carlo integración y Monte Carlo Mark Chain. Hablamos de Monte Carlo porque usamos simulaciones o procesos estocásticos para obtener información.

a) Integración Monte Carlo

Estamos en el caso dónde:

$$l(x,y,z) = 1$$

En este caso $I = V \iiint g(x,y,z)\frac{1}{V}dxdydz$

Con $V = \iiint 1\ dxdydz$. V representa el volumen del conjunto E.

En consecuencia $p(x,y,z) = \frac{1}{V}$ es la densidad de la ley uniforme sobre E

Suponemos que tenemos (x1,y1,z1) ,(x2,y2,z2),….,(xn,yn,zn) una muestra independiente de la variable uniforme sobre E. En consecuencia:

$$\frac{V}{n}\sum_{i=1}^{n} g(x_i,y_i,z_i) \approx \iiint g(x,y,z)dxdydz$$

Simular la ley uniforme [0,1] no es difícil porque existen varios algoritmos en librerías básicas para su simulación y sirven como base para simular otras probabilidades numéricamente. Vamos a estudiar las simulaciones en la parte 2). Además si sabemos simular una ley uniforme [0,1] unidimensional podemos rápidamente pasar a la simulación de una ley uniforme multidimensional [ax,bx][ay,by][az,bz].

Intervalo de confianza:

Además sabemos con la ley de los grandes números que $\frac{V}{n}\sum_{i=1}^{n} g(X_i,Y_i,Z_i)$ con n>=30 sigue un ley normal $N\left(I,\widehat{\sigma^2}\right)$

Con $\widehat{\sigma^2} = \frac{1}{n-1}\sum_{i=1}^{n}(Vg\left(x_i,y_i,z_i\right) - I)^2$ la varianza estimada corregida

En consecuencia un intervalo de confianza a 0.95 de I puede ser estimado con :

$$\left[I - \frac{1.96 * \sqrt{\widehat{\sigma^2}}}{\sqrt{n}}, I + \frac{1.96 * \sqrt{\widehat{\sigma^2}}}{\sqrt{n}}\right]$$

b) MCMC:

Estamos en el caso donde $l(x,y,z)$ es más complejo que el caso a).Entonces tenemos una densidad p(x,y,z) más complejo. Aplicamos los mismos resultados que en el caso general. La dificultad aquí es simular la ley de probabilidad de densidad $p(x,y,z)$ sobre E. Para tener una estimación de la muestra de p utilizamos una versión del algoritmo MCMC: Hastings-Metrópolis

Nuestros puntos iniciales:$x_o y_o z_o$

Generamos $x_1 y_1 z_1$ con una simulacion de la ley de probabilidad normal N 3D (en este caso) de promedio $x_o y_o z_o$ y de matriz de convarianza $\begin{pmatrix} \sigma^2 & 0 & 0 \\ 0 & \sigma^2 & 0 \\ 0 & 0 & \sigma^2 \end{pmatrix}$, esta matriz de convarianza impliqua que las variables marginales son independientes. El σ^2 representa el tamaño brinco, si es demasiado grande o pequeño la cadena no va a converger porque en el primer caso podría generar grandes variaciones en la simulación y en el segundo caso podría hacer pequeñas variaciones en la simulación. Gelman-Rubin proponen por experiencia un tamaño brinco igual a $2.4^2 * \widehat{\sigma^2}$ con la estimación de la varianza de los primeros casos. Aquí el tamaño brinco optimo depende de la distribución de p. Para simular nuevos candidatos podemos elegir una otra probabilidad que la ley normal.

Guardamos en el vector simulación $x_1 y_1 z_1$ si $\frac{p(x_1,y_1,z_1)}{p(x_0,y_0,z_0)} > U$ con U~Uniform([0,1])

Si no guardamos $x_1 y_1 z_1$, empezamos de nuevo para generar $x_1 y_1 z_1$.

Cuando tenemos $x_1 y_1 z_1$hacemos el mismo proceso pare generar y guardar $x_2 y_2 z_2$ en una vector simulación.

La dificultad aquí es construir un simulador de una ley normal multidimensional y elegir un buen tamaño brinco.

c) Box Muller

Para la implementación de una ley normal multidimensional se suele usar la transformación Box-Müller, que consiste en hacer un cambio de coordenadas polares para lograr obtener una expresión que sí tenga una forma analítica cerrada. Una vez obtenida una simulación invariada. Para el caso multivariado el siguiente reto es el de calcular la respectiva matriz de covarianzas.

2) Simulación de números aleatorios

La generación de números aleatorios es ampliamente utilizada en ramas como las ciencias de la computación e ingeniera. La aleatoriedad suele estar presente en la formulación del problema como lo es el ruido u otras perturbaciones aleatorias. Además, muchos algoritmos probabilistas parten del hecho que se cuenta con variables aleatorias, por ejemplo, simulación de Monte Carlo y técnicas de optimización estocástica o algoritmos genéticos. Dado que los números "aleatorios" que se calculan son deterministas por la realización de algoritmos, no son realmente aleatorios, se les conoce más apropiadamente como pseudo-aleatorio. Si se requiere una gran cantidad de números aleatorios, como en el caso de la simulación de Monte Carlo, la calidad del generador de números aleatorios puede resultar relevante.

En la práctica resulta imposible generar números aleatorios perfectos, sin embargo la siguiente lista enumera el comportamiento ideal:

1. Se distribuyen uniformemente

2. No están correlacionados

3. Nunca se repiten

4. Satisfacer cualquier prueba estadística de aleatoriedad

5. Son reproducibles (para propósitos de depuración)

6. Son portátiles (lo mismo en cualquier computadora)

7. Se puede cambiar ajustando un valor (seed inicial)

8. Puede dividirse fácilmente en muchas subsecuencias independientes

9. Puede generarse rápidamente usando memoria limitada de la computadora

Generador lineal congruencial:

Lo que define este generador es la relación de recurrencia:

$$X_{n+1} = (aX_n + c)\ mod\ m$$

$$X_0, X_1, \dots, X_n\ es\ la\ sequencia\ de\ los\ valores\ pseudo - aleatorio$$

$$m\ es\ el\ modulo: 0 < m$$

$$a\ es\ el\ multiplicator: 0 < a < m\ el$$

$$c\ es\ el\ increment: 0 \leq c < m$$

$$X_0\ es\ la\ semilla\ , 0 \leq X_0 < m$$

El glibc utilizado por el compilador gcc cuando llamamos rand() es el *Linear congruential generator* .

Glibc tiene como valores para el generador lineal congruencial:

$$m = 2^{31} - 1 = 2\ 147\ 483\ 647\ (es\ el\ RAND\_MAX())$$

$$a = 1103515245$$

$$c = 12345$$

$$X_0\ corresponde\ a\ la\ valor\ dado\ en\ el\ srand()$$

Para cada semilla, se genera una lista de números entre 0 y m. Esta lista puede verse como pseudo-aleatoria, porque con la multiplicación, adición y modulo es difícil de ver la relación que liga estos números. Estas 3 operaciones permiten visitar números con escalas diferentes para una simulación completa sobre el intervalo [0,m]. Además tenemos m semillas posibles así en el caso de rand() de c tenemos 2 147 483 647 listas posibles de números "con poco sentido entre ellos" que llamamos números pseudo-aleatorios.

LCG es rápido y no necesita una memoria grande. Esto permite de simular fácilmente LCG en diferentes threads o procesos en una misma máquina. Sin embargo existe una importante correlación entre los diferentes X de una misma simulación .Entonces LCG tiene una capacitad de aleatoriedad baja lo que es problemático para MC integración. Sin embargo como hacemos simulaciones en diferentes procesos con semillas diferentes eso nos da al final una muestra más aleatoria (aproximadamente 10 procesos).

III) Implementación de integración MC en MPI

Nuestro objetivo aquí es implementar el algoritmo de MC integración en un sistema de memoria distribuida por el intermedio de MPI para aumentar el número de simulaciones con un bueno tiempo de computación y tener una muestra más aleatoria. En primera parte vamos a introducir como creamos nuestra ambiente pseudo-distribuida .En segunda parte presentamos nuestra función unif y cómo utilizarla con los diferentes procesos. En tercera parte explicamos cómo dividimos las tareas entre los procesos y cómo funciona la comunicación entre ellos. Al final describimos como utilizar el algoritmo e interpretamos sus resultados.

1) Cluster MPI

Aprovechamos del trabajo de Nikyle Nguyen en licence MIT disponible en github que es el autor del libro *Distributed MPI cluster with Docker Swarm mode* . Con su trabajo utilizamos un docker-compose para levantar un número importante de contenedores (10, 15, 20) que están ligados de manera automático por la utilización del docker-compose. Pusimos un tutorial en la misma carpeta del documento para poder levantar un cluster de 10 contenedores y aplicar MPI con nuestro algoritmo de integración de Monte Carlo.

2) Función unif()

Construimos nuestra función unif() como:

```
double unif(double a, double b) {

   return ( rand()/(double)RAND_MAX ) * (b-a) + a ;

}
```

Vimos que rand() genera un número aleatorio entero entre 0 y RAND_MAX de manera pseudo-aleatoria. El RAND_MAX en mi máquina es igual a 2 147 483 647 ($2^{31} - 1$).

De hecho, rand() devuelve un número extraída de una secuencia de números pseudo-aleatorios generados por el algoritmo dado un numero inicial que es la semilla. El número fue generado de manera pseudo-aleatoria pero necesitamos empezar a cada vez en un nuevo punto cuando utilizamos rand() en cada proceso y cuando lanzamos el algoritmo en el objetivo de no repetir la secuencia entre los procesos y entre cada llamada del algoritmo. La función srand(semilla) permite de inicializar la semilla antes de llamar rand().

En consecuencia utilizamos una variable semilla_local tal que después MPI_Init(&argc,&argv) : semilla_local=time(NULL)+100*rango_processo

En la funcion simul_local (la función que hace la estimación local del integración para cada proceso) llamamos srand(semilla_local).

Utilizamos time(NULL) dentro srand() que da el número de segundo desde 1970. Este valor sirve a que cada nueva llamada del algoritmo no simulemos los mismos números.

Para que un proceso llamando rand() dentro la funcion simul_local pueda leer una serie de números diferentes de los otros procesos utilizamos 100*rango_processo. El rango_processo es el rango del proceso, es el número que le caracteriza dentro el conmutador MPI_COMM_WORLD. Además pusimos 100 como multiplicador para evitar la siguiente situación: si el proceso 1 se ejecuta un segundo después del proceso 2 va a tener la misma semilla. Pusimos antes N*rango_processo con N número de simulaciones pero damos cuenta por ejemplo que si N es más grande que 1 000 000 000 vamos a iniciar una semilla más grande que RAND_MAX() (el modulo m) para los procesos de rango más de 1

En consecuencia con el hecho que cada proceso define time(NULL)+100*rango_processo como semilla para el rand() aproximamos la independencia de las simulaciones entre los proceso y entre cada llamada del algoritmo.

rand()/(double)RAND_MAX genera un numero aleatoria double entre 0 y 1 y pusimos /(double) para tener un división real.

(rand()/(double)RAND_MAX)*(b-a) + a genera un número aleatorio entre a y b. Sin embargo no generamos todas los valores entre a y b porque la distancia más pequeña entre 2 números sacados será de (b-a)/RAND_MAX. En general b-a << 2 147 483 647 (RAND_MAX).Sin embargo si no es el caso debemos cambiar de métodos para generar números aleatorios.

3) Paralización y comunicación entre los procesos.

a) Paralización del algoritmo MC integración

Para aprovechar de nuestro cluster de docker en nuestra maquina utilizamos MPI (message passing librerary interface) que van a permitir a cada proceso de un contenedor de ejecutar algoritmos en paralelo y comunicarse entre ellos.

Cada proceso va a ejecutar el mismo algoritmo de simulación de Monte Carlo que va a permitir estimar en cada proceso el integral I con un número de simulaciones idéntico N. Al final, todos los procesos de rango diferente de 0 envían sus resultados al proceso 0 que hace el promedio de la estimación de I de cada estimación local. Además cada proceso hace también la simulación de $E(g(X)^2)$ y envía su resultado al proceso 0 que va estimar la varianza de Vg(X) para dar un intervalo de confianza.

El algoritmo de simulación local de Monte Carlo:

```
double simulacion_local(double a_x,double b_x,double a_y,double b_y,double a_z,double b_z,int N,int raiz,int choice){
        int i=0;
        double sim_x;
        double sim_y;
        double sim_z;
        double sim_mean;
        double sim_for_var;
        srand(raiz);
        for (i=0;i<N;i++){
                sim_x=unif(a_x,b_x);
                sim_y=unif(a_y,b_y);
                sim_z=unif(a_z,b_z);
                if (choice==0) { sim_mean+=f(sim_x,sim_y,sim_z); }
                else { sim_for_var+=pow(f(sim_x,sim_y,sim_z),2);}
        }
        if (choice==0){ return (b_x-a_x)*(b_y-a_y)*(b_z-a_z)*(1/(double)N)*sim_mean          }
        else {return pow((b_x-a_x)*(b_y-a_y)*(b_z-a_z),2)*(1/(double)N)*sim_for_var;}}
```

Aquí es el lugar donde los procesos llaman la función simulacion_local:

```
MPI_Init(&argc,&argv);

start = MPI_Wtime();//Te da el tiempo en segundo cuando un processo esta llamado porque hay MPI_Init() antes.

MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

MPI_Comm_rank(MPI_COMM_WORLD, &rango_processo);

semilla_local=time(NULL)+N*rango_processo;

simul_local[0]=simulacion_local(a_x,b_x,a_y,b_y,a_z,b_z,N,semilla _local,0);

simul_local[1]=simulacion_local(a_x,b_x,a_y,b_y,a_z,b_z,N,semilla _local,1);
```

Comentarios:

Podemos observar que los procesos llaman a simulacion_local 2 veces porque hacen 2 simulaciones distintas una para estimar la integral I y el otro para estimar más tarde en el proceso 0, la varianza. Además podemos ver que cada proceso tiene una semilla diferente para generar números diferentes entre proceso. Observamos que para la llamada de las 2 simulacion_local conservan la misma semilla. En efecto es importante que en los 2 simulaciones consecutivas tienen el mismo conjunto de números pseudo-aleatorios para que pueden estimar la varianza de la primer simulación y no de otra. Hicimos esto porque no podemos regresar 2 resultados numéricos en una misma llamada de una función en C. Pues hay un último parámetro choice que permite de elegir cual es el resultado que queremos con simul_local.

b) Comunicación entre los procesos

Cada proceso del conmutador MPI_COMM_WORLD diferente de 0 envía un array que tiene la estimación local de I y la estimación $E(g(X)^2)$ local.

```
double simul_local[2];

……..

raiz_local=time(NULL)+N*rango_processo;

simul_local[0]=simulacion_local(a_x,b_x,a_y,b_y,a_z,b_z,N,raiz_local,0);

simul_local[1]=simulacion_local(a_x,b_x,a_y,b_y,a_z,b_z,N,raiz_local,1);

if(rango_processo!=0){

MPI_Send(&simul_local,2,MPI_DOUBLE,0,1,MPI_COMM_WORLD);}
```

El proceso 0 es el proceso maestro que va recibir y agrupar todas las estimaciones locales de cada proceso para dar la estimación final del integral y sacar el intervalo de confianza sobre la estimación de I.

```
else{

simul_general_mean+=simul_local[0];

simul_general_for_var+=simul_local[1];


for(num_proceso=1;num_proceso<comm_sz;num_proceso++){

MPI_Recv(&simul_local,2, MPI_DOUBLE, num_proceso,1,
MPI_COMM_WORLD,MPI_STATUS_IGNORE);

simul_general_mean+=simul_local[0];

simul_general_for_var+=simul_local[1];

        }

}
```

Comentarios:

Cada proceso tiene un lugar en su memoria (simul_local) donde almacena el resultado de la función simulacion_local(...) y el proceso 0 va a agregar progresivamente los resultados de todos los procesos en las variables simul_general_mean y simul_general_for_var. Sin embargo el proceso va a recibir los datos de los otros procesos en la misma dirección simul_local de su memoria. Es por eso que antes que su variable simul_local cambia, guardamos el resultado de la llamada de la función simul_local por el proceso 0.

Presentamos aquí el último paso del proceso 0 cuando recibió todos los mensajes de los procesos y determina la desviación estándar.

```
if (rango_processo==0){

        simul_general_mean=(1/(double)comm_sz)*simul_general_mean;

        desv_estandar=sqrt((1/(double)(comm_sz))*simul_general_for_var-
pow(simul_general_mean,2));


        printf("\n\n");

        printf("Integral sobre [%1.2f,%1.2f][%1.2f,%1.2f][%1.2f,%1.2f] de log(x)-y^2+z: %1.8f\n\n
",a_x,b_x,a_y,b_y,a_z,b_z,simul_general_mean);

        ……..

}
```

4) Estimación del intervalo de confianza

Vamos a explicar cómo determinamos la desviación estándar de I y cuál fue la dificultad para estimarla. En general para estimar la varianza de una muestra necesitamos guardar todas las simulaciones hechas en todos los procesos después de haber estimado el integral I. En efecto la estimación de la varianza no corregida para un proceso es:

$$\widehat{\sigma^2} = \frac{1}{n}\sum_{i=1}^{n}(Vg(x_i,y_i,z_i) - I)^2$$

Sin embargo con : $I = \frac{1}{n}\sum_{i=1}^{n} Vg(x_i y_i z_i)$

Entonces podemos transformar esta relación:

$$\widehat{\sigma^2} = \frac{1}{n}\sum_{i=1}^{n} V^2 g(x_i,y_i,z_i)^2 - I^2$$

$\frac{1}{n}\sum_{i=1}^{n} V^2 g(x_i,y_i,z_i)^2 = simul\_local[1]$ corresponde al segundo resultado de la función simul_local

Entonces:

$$Varianza\_estimada = \frac{1}{n\_procesos} * \left(\sum_{i=1}^{n\_processos} simul\_local[1]\_proceso\_i \right) - I^2$$

$$Con\ \ I = \frac{1}{n\_procesos} * \left(\sum_{i=1}^{n\_processos} simul\_local[0]\_proceso\_i \right)$$

5) Resultados e interpretación

En el algoritmo definimos la función a integrar:

double f(double x, double y,double z) { return log(x)-pow(y,2)+z; }

Si queremos cambiar de función en 3 dimensiones necesitamos cambiar esta parte del código.

En el nodo maestro del cluster tenemos que llamar al algoritmo como:

mpirun -n 10 ./MC_integration_MPI_var.out N ax bx ay by cx bx

- N:número de simulaciones idéntica para todos los procesos
- ax,bx: el tamaño de integración de f para la variable x.
- ay,by: el tamaño de integración de f para la variable y.
- az,bz: el tamaño de integración de f para la variable z.

Ejemplos:

En este caso vamos a hacer la integración de la función $f(x, y, z) = \log(x) - y^2 + z$

Caso: 10 contenedores

a) Fijamos los valores de ax,bx,ay,by,az,bz y cambiamos el número de simulaciones:

- **/project $ mpirun -n 10 ./mc_integration_MPI_var 10000 1 5 4 10 10 20**

Integral sobre [1.00,5.00][4.00,10.00][10.00,20.00] de log(x)-y^2+z: -8625.46493794

Interval de confianza de 95/100 es:
[-8740.91771736,-8510.01215852]

Numero de simulaciones realizados de manera pseudo-independiente en el cluster: 100000
Tiempo de ejecucion = 0.096872 s

- **/project $ mpirun -n 10 ./mc_integration_MPI_var 100000 1 5 4 10 10 20**

Integral sobre [1.00,5.00][4.00,10.00][10.00,20.00] de log(x)-y^2+z: -8627.54938163

Interval de confianza de 95/100 es:
[-8664.10597073,-8590.99279253]

Numero de simulaciones realizados de manera pseudo-independiente en el cluster: 1000000

Tiempo de ejecucion = 0.274369 s

- **/project $ mpirun -n 10 ./mc_integration_MPI_var 1000000 1 5 4 10 10 20**

Integral sobre [1.00,5.00][4.00,10.00][10.00,20.00] de log(x)-y^2+z: -8637.17982813

Interval de confianza de 95/100 es:
[-8648.73438708,-8625.62526918]

Numero de simulaciones realizados de manera pseudo-independiente en el cluster: 10000000

Tiempo de ejecucion = 1.528732 s

- **/project $ mpirun -n 10 ./mc_integration_MPI_var 10000000 1 5 4 10 10 20**

Integral sobre [1.00,5.00][4.00,10.00][10.00,20.00] de log(x)-y^2+z: -8636.85258686

Interval de confianza de 95/100 es:
[-8640.50762941,-8633.19754432]

Numero de simulaciones realizados de manera pseudo-independiente en el cluster: 100000000

Tiempo de ejecucion = 14.292012 s

- **/project $ mpirun -n 10 ./mc_integration_MPI_var 100000000 1 5 4 10 10 20**

Integral sobre [1.00,5.00][4.00,10.00][10.00,20.00] de log(x)-y^2+z: -8637.17886571

Interval de confianza de 95/100 es:
[-8638.33468286,-8636.02304856]

Numero de simulaciones realizados de manera pseudo-independiente en el cluster: 1000000000

Tiempo de ejecucion = 142.423883 s

Interpretacion:

Observamos que el intervalo de confianza disminuye cuando el número de simulaciones N crece. Los tiempos de cómputo son buenos. En efecto desde el caso N=10 000 000 tenemos un intervalo de confianza correcto al nivel del cuarta cifra significativa y simulamos 100 000 000 números pseudo-aleatorios en 14.5 segundos. Además después el caso N=10 000 000 el algoritmo varía entre 8636 y 8637

b) Fijamos el tamaño de los intervalos y el número de simulación y cambiamos el número de procesos

- **/project $ mpirun -n 10 ./mc_integration_MPI_var 10000 1 5 4 10 10 20**

Integral sobre [1.00,5.00][4.00,10.00][10.00,20.00] de log(x)-y^2+z: -8625.46493794

Interval de confianza de 95/100 es:

[-8740.91771736,-8510.01215852]

Numero de simulaciones realizados de manera pseudo-independiente en el cluster: 100000

Tiempo de ejecucion = 0.096872 s

- **/project $ mpirun -n 20 ./mc_integration_MPI_var 10000 1 5 4 10 10 20**

Integral sobre [1.00,5.00][4.00,10.00][10.00,20.00] de log(x)-y^2+z: -8643.48938492

Interval de confianza de 95/100 es:

[-8758.87944451,-8528.09932532]

Numero de simulaciones realizados de manera pseudo-independiente en el cluster: 200000

Tiempo de ejecucion = 0.347049 s

Interpretacion:

Aquí el tiempo de cómputo aumenta 3 veces dado el caso de 10 procesos. Es normal, estamos en el caso de un cluster de 10 contenedores y con 20 procesos. Significa que hay procesos que esperan que los primeros se terminen. Además este tiempo se justifica también porque el proceso 0 espera el mensaje del procese1 y del proceso2 hasta el proceso 20.Sin embargo el proceso 20 puede acabar antes el processo1 y el processo2 puede esperar que los primeros procesos se terminan.

Caso: 20 contenedores

a) Comparamos con un caso anterior:

- **/project $ mpirun -n 10 ./mc_integration_MPI_var 10000000 1 5 4 10 10 20**

Integral sobre [1.00,5.00][4.00,10.00][10.00,20.00] de log(x)-y^2+z: -8637.73896661

Interval de confianza de 95/100 es:

[-8641.39398627,-8634.08394696]

Numero de simulaciones realizados de manera pseudo-independiente en el cluster: 100000000

Tiempo de ejecucion = 14.055292 s

Interpretación:

El caso anterior tenía como intervalo de confianza: [-8640.50762941,-8633.19754432] y un tiempo de ejecución: 14.29s .Nada ha cambiado es normal porque 10 contenedores de 20 fueron asignados para tratar los 10 procesos.

- **/project $ mpirun -n 20 ./mc_integration_MPI_var 10000000 1 5 4 10 10 20**

Integral sobre [1.00,5.00][4.00,10.00][10.00,20.00] de log(x)-y^2+z: -8637.15502742

Interval de confianza de 95/100 es:
[-8640.81001696,-8633.50003788]

Numero de simulaciones realizados de manera pseudo-independiente en el cluster: 200000000

Tiempo de ejecucion = 24.023590 s

Interpretación:

Aquí el tiempo no fue doblado pero aumenta comparado al mismo caso que con 10 procesos. Eso muestra la limitación de mi ordenador para manejar 20 contenedores. En efecto, tenemos 20 contenedores y hay algún proceso de los 20 que espera.

b) Aumentamos el tamaño de los intervalos de integración

Ahora estamos interesados en ver como la precisión disminuye cuando aumentamos el rango de los intervalos.

- **/project $ mpirun -n 10 ./mc_integration_MPI_var 10000000 1 50 4 10 10 100**

Integral sobre [1.00,50.00][4.00,10.00][10.00,100.00] de log(x)-y^2+z: 158703.52620907

Interval de confianza de 95/100 es:

[158118.83880278,159288.21361536]

Numero de simulaciones realizados de manera pseudo-independiente en el cluster: 100000000

Tiempo de ejecucion = 12.091890 s

- **/project $ mpirun -n 10 ./mc_integration_MPI_var 100000000 1 50 4 10 10 100**

Integral sobre [1.00,50.00][4.00,10.00][10.00,100.00] de log(x)-y^2+z: 158597.52968951

Interval de confianza de 95/100 es:

[158412.63798526,158782.42139376]

Numero de simulaciones realizados de manera pseudo-independiente en el cluster: 1000000000

Tiempo de ejecucion = 117.366433 s

- **/project $ mpirun -n 10 ./mc_integration_MPI_var 1000000000 1 50 4 10 10 100**

Integral sobre [1.00,50.00][4.00,10.00][10.00,100.00] de log(x)-y^2+z: 158553.11208230

Interval de confianza de 95/100 es:

[158494.64709669,158611.57706790]

Numero de simulaciones realizados de manera pseudo-independiente en el cluster: 1410065408

Tiempo de ejecucion = 1189.691078 s

- **/project $ mpirun -n 20 ./mc_integration_MPI_var 100000000 1 50 4 10 10 100**

Integral sobre [1.00,50.00][4.00,10.00][10.00,100.00] de log(x)-y^2+z: 158559.55012562

Interval de confianza de 95/100 es:

[158374.66385752,158744.43639372]

Numero de simulaciones realizados de manera pseudo-independiente en el cluster: 2000000000

Tiempo de ejecucion = 264.352194 s

Interpretación:

Cuando $N \leq 10\,000\,000$ tenemos una precisión con 2 cifras significativas, no es tan bueno pero el algoritmo se ejecutó en 16 segundos. Tenemos una precisión al nivel de la tercera cifra significativa con $N \geq 100\,000\,000$. Sin embargo no logramos a tener una precisión al nivel del 4 cifra significativa con N=1 000 000 000 (lo que significa que con 10 procesos el algoritmo va a hacer 10 000 000 000 de simulaciones).Tenemos la impresión que después una N grande dada, la precisión no aumenta de manera importante como si la precisión siguiera un ley logarítmica según N. Además tenemos para el caso 1 000 000 000 y 10 procesos:

Numero de simulaciones realizadas de manera pseudo-independiente en el cluster: 1410065408

Normalmente deberemos tener 10 000 000 000. Además pensamos que el código ejecuta las 10 000 000 000 simulaciones porque el tiempo de ejecución fue de más de 1000 segundos. En consecuencia creamos esta problema se origina desde la función printf().

Conclusión:

El método presentado de integración Monte Carlo arrojó muy buenos resultados en cuanto al performance del procesamiento. Por ejemplo, con 100 000 000 simulaciones el proceso tardó 11.465 s.

Respecto a la calidad de la integración, es importante señalar que al simular la unif(a,b), dado que se hace mediante una transformación, la distancia mínima entre 2 simulaciones es (b-a)/RAND_MAX. Además la técnica LCG utilizado por rand() no es la mejor al nivel de calidad de aleatoriedad. Adicionalmente la literatura recomienda el uso de distintos generadores de números aleatorios para tener una protección más contra posibles correlaciones entre las diversas simulaciones generadas.

Una gran mejora a nuestro código debería hacer una segunda capa de paralización. Nos gustaría usar los procesos para dividir los intervalos de integración y los threads para aumentar el número de simulaciones dentro un proceso.

Referencias:

W. Petersen, P. Arbenz Introduction to Parallel Computing: A Practical Guide with Examples inC
https://www.researchgate.net/publication/268617797_Introduction_to_parallel_computing_A_practical_guide_with_examples_in_C)

P. D. Coddington. Random Number Generators for Parallel Computers
http://surface.syr.edu/cgi/viewcontent.cgi?article=1012&context=npac

M. Mascagni. Parallel Pseudorandom Number Generation
http://www.cs.fsu.edu/~mascagni/papers/IIP1.pdf

R. P. Brent. Fast and Reliable Random Number Generators for Scientific
Computing](http://maths-people.anu.edu.au/~brent/pd/rpb217a.pdf)

R. P. Brent. Random Number Generation and Simulation on Vector and Parallel
Computers](http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.2.8988&rep=rep1&type=pdf)

Metropolis–Hastings algorithm
wikipedia](https://en.wikipedia.org/wiki/Metropolis%E2%80%93Hastings_algorithm)

Curso de Mauricio Garcia Tec Estadistica Computacional
https://1drv.ms/b/s!AtYhTtvZ1LiEg9ZfLV-dAw_uLrInpw

OpenMP básico
http://www.cs.cornell.edu/~bindel/class/cs5220-s10/slides/lec06.pdf

L'aleatoire en C et C++ con rand()
https://openclassrooms.com/courses/l-aleatoire-en-c-et-c-se-servir-de-rand-1

Using Docker container Alpine-MPICH to develop MPI

https://asciinema.org/a/93067

Github associada

(https://github.com/NLKNguyen/alpine-mpich)