COMPARACIÓN DEL CÁLCULO DE PI MEDIANTE DIFERENTES MÉTODOS DE APROXIMACIÓN

RAMIRO ANDRÉS BARRIOS VALENCIA

DANIEL BENAVIDES SARAY
FERNAN ALBERTO CAÑAS
EDGAR MARTINEZ LOAIZA
ALEJANDRO PLAZA PARRA

UNIVERSIDAD TECNOLÓGICA DE PEREIRA
PEREIRA, RISARALDA
25 DE JUNIO DE 2019

INTRODUCCIÓN

El número pi

"La historia de PI refleja el más influyente, el más grave y, a veces, el tonto aspecto de las matemáticas. Una sorprendente cantidad de los más importantes matemáticos han contribuido a su evolución, directa o indirecta.
Pi es uno de los pocos conceptos en las matemáticas, cuya mención evoca una respuesta de reconocimiento y el interés en aquellos que no se traten profesionalmente con el tema. Ha sido una parte de la cultura humana y la imaginación, estudiado durante más de veinticinco siglos.
El cálculo de Pi es prácticamente el único tema de los más antiguos estratos de las matemáticas que es aún de gran interés para la investigación matemática moderna."

¿Que es el número PI?

π (pi) es la relación entre el perímetro de una circunferencia y la longitud de su diámetro, no es un número exacto, pertenece al conjunto de números irracionales, es decir, que tiene infinitos números decimales.

Pero, cómo podemos hallar pi desde una base matemática, para esto vamos a estudiar dos maneras de hallarlo.

- Método Monte-carlo dartboard
- Método aguja de Buffon

MÉTODO MONTECARLO DARTBOARD

¿Qué es el Método Monte Carlo?

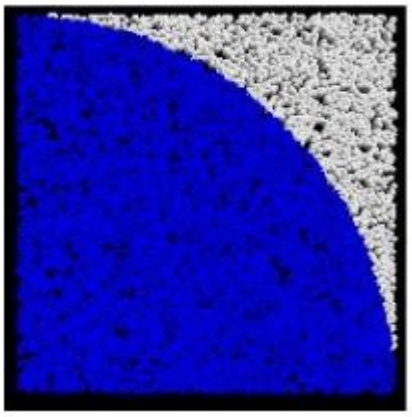
El método Monte Carlo es un método en el que por medio de la estadística y la probabilidad podemos determinar valores o soluciones de ecuaciones que calculados con exactitud son muy complejas, pero que mediante este método resulta sencillo calcular una aproximación al resultado que buscamos.

El método Monte Carlo fue desarrollado en 1944 en Laboratorio Nacional de Los Álamos, como parte de los estudios que condujeron al desarrollo de la bomba atómica. En un principio lo desarrollaron los matemáticos John Von Neumann y Stanislaw Ulam aunque fueron otros matemáticos quienes con su trabajo le dieron una solidez científica, Harris y Herman Kahn.

La idea le surgió a Ulam, mientras jugaba a las cartas. Se le ocurrió un método en el que mediante la generación de números aleatorios, pudieran determinar soluciones a ecuaciones complejas que se aplican en el estudio de los neutrones. Era como generar los números con la ayuda de una ruleta, de ahí su nombre.

Calculando el valor de Pi, con la ayuda del Método Monte Carlo.

Para entender este método imaginemos que tenemos un círculo en el interior de un cuadrado y que de alguna manera podemos dibujar puntos aleatoriamente en toda la figura , una vez tengamos suficientes puntos marcados en la figura notaremos que algunos quedan dentro del circulo y otros no, usaremos esta información para encontrar una aproximación de pi de la siguiente manera:



Supongamos que tenemos un cuadrado con lado: 2r lo que nos daría que el círculo interior a su vez cuente con un radio: r, el área del cuadrado y del círculo resultantes serían entonces:

$$A_{cuadrado} = (2r)^2 = 4r^2$$

$$A_{circulo} = \pi r^2$$

Si dividimos estos valores entre ellos tendremos entonces:

$$\frac{A_{circulo}}{A_{cuadrado}} = \frac{\pi r^2}{4r^2} = \frac{\pi}{4}$$

si dividimos el valor total de puntos generados en el área del cuadrado entre los que se encuentran dentro del área del círculo podemos encontrar un valor aproximado de pi

$$\frac{Area - del - circulo}{Area - del - cuadrado} = \frac{Numero - de - puntos - dentro - del - circulo}{Numero - de - puntos - totales}$$

$$\frac{puntos\ del\ circulo}{puntos\ totales} \approx \frac{\pi}{4}$$

De esta manera queda mucho más sencillo diseñar un algoritmo que lo resuelva por nosotros cuantas veces queramos.
Hemos diseñado entonces el siguiente programa para que realice esta ardua tarea por nosotros, el cual explicamos y enseñamos a continuación:

```
void MontecarloS(int const &n, double &pi)
{
    std::random_device rd;
    std::mt19937 mt(rd());
    std::uniform_real_distribution<double> dist(0, 1);

    int PuntosCirculo = 0;
    double x, y;
    for(int i = 0; i <= n; i++)
    {
        x = dist(mt);
        y = dist(mt);

        if (sqrt((x*x)+(y*y)) <= 1)
          PuntosCirculo++;
    }
    pi = 4.0*((double)PuntosCirculo/(double)(n));
}
 clock_t StartingPoint = clock();
 MontecarloS(s, pi);
 StartingPoint = clock() - StartingPoint;

 printf("\nAproximación de pi: %f \n\n", pi);
 printf("(%f segundos).\n",((float)StartingPoint)/CLOCKS_PER_SEC);
```

1. El programa está realizado en c++, lo primero es definir e importar las librerías que necesitaremos, en este caso <math.h><iostream><stdlib.h><random><stdio.h>.

2. Cómo trabajaremos sobre números creados al azar, lo primero es definir el rango en el que estos números serán creados, como utilizaremos un círculo de radio=1 es indispensable crear estas variables (x,y) como tipo doble, para el manejo de cifras significativas.

3. La manera más sencilla para saber si el punto generado esta dentro del círculo es saber su distancia respecto al punto de origen de este, así que emplearemos el teorema de pitágoras para saber si esta es menor o igual a 1, es decir que el punto se encuentra dentro de los límites.

4. En este caso el tener que llamarlo desde el main implica que nuestro contador de tiempo será iniciado y finalizado allí mismo, lo ideal es que comience a correr una vez que se conoce el valor de n (número de puntos totales), con esto sabremos cuánto tarda la máquina en encontrar esta aproximación, lo que nos sera muy util mas adelante

5. inicializamos la variable PuntosCirculo en 0, es esta quién irá llevando la cuenta de el número de puntos que cumplen con las condiciones de distancia<=1, para que esto funcione, necesitamos entonces la creación de un ciclo for que genere la pareja de coordenadas y evalúe que cumplan la condición y vaya sumando y almacenando el número de puntos que si lo hacen en esta variable PuntosCirculo.

6. Una vez que se haya recorrido el for el número de veces solicitado y almacenado el número de aciertos dentro del círculo procederemos a hallar el valor de pi con la fórmula que encontramos al realizar el análisis del algoritmo anteriormente.

$$\frac{puntos\ del\ circulo}{puntos\ totales} \approx \frac{\pi}{4}$$

7. Ya solo nos queda imprimir el valor encontrado de la aproximación de pi y el tiempo que tomó encontrarlo.

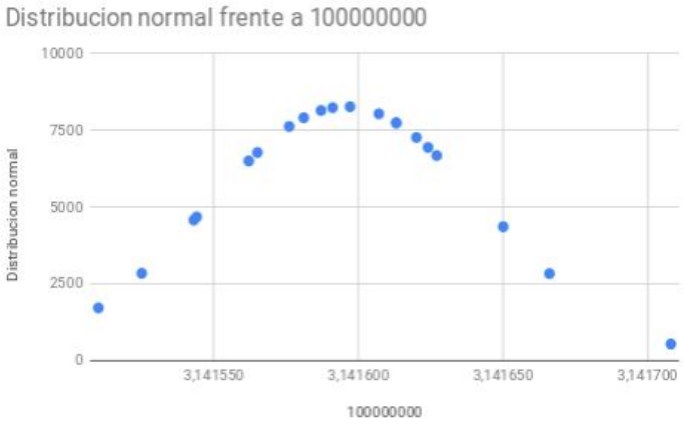
CONFIABILIDAD DE LOS DATOS:

Esta confiabilidad nos va a permitir saber y estar seguros de que los datos obtenidos convergen al valor de pi que conocemos.

Ya que estas pruebas generan números aleatorios constantemente el valor de los resultados tiende a variar un poco entre ellos cada vez que corremos el programa, pero podemos encontrar un valor bastante fiable entre más iteraciones realicemos y entre más datos tomemos, por ello hemos escogido realizar recolectar 20 datos con 100.000.000 iteraciones lo que tardo en total 7.1min en devolver los valores solicitados.

para este reto hemos definido que al menos 4 dígitos del valor de la media encontrada coincidan con el valor conocido de pi. Construimos una tabla con los valores encontrados, con su respectiva media, y desviación estándar, con estos datos podemos graficar su respectiva campana de gauss, lo que nos permitirá conocer la distribución de los resultados.

Datos	1,000,000,000	Distribucion normal	Tiempo(seg)
1	3,141613	7743,564565	192,298935
2	3,141576	7627,370341	192,320236
3	3,141708	542,6893505	192,603668
4	3,141613	7743,564565	194,688339
5	3,141562	6504,322331	191,72496
6	3,141510	1720,768193	192,514755
7	3,141650	4362,89824	192,576385
8	3,141543	4578,911901	194,118668
9	3,141525	2845,382144	192,983841
10	3,141607	8039,864657	192,910278
11	3,141591	8238,707631	191,607574
12	3,141544	4682,379949	193,317078
13	3,141565	6778,085243	192,026459
14	3,141666	2836,76688	192,824799
15	3,141624	6943,481427	192,239532
16	3,141581	7910,54313	194,855255
17	3,141587	8147,785736	195,51712
18	3,141620	7268,012239	193,40654
19	3,141627	6679,39425	192,634079
20	3,141597	8269,595531	192,13063
Media =	3,141595	----------	192,9649566
Desviación =	,0000482171349	----------	1,06785485
Tiempo total en seg =			3859,299131
Tiempo total en min =			64,32165218



Con esta representación de datos nos damos cuenta que el método aunque sencillo tarda casi medio segundo en arrojar el resultado con una precisión bastante alta a los 4 dígitos que teníamos previstos acertar, la distribución nos dice que si ponemos más iteraciones están tenderán cada vez más al valor real conocido (3.141595....)

Las especificaciones de la máquina donde fue ejecutado son las siguientes:

Máquina: Lenovo IdeaPad S400 Touch
CPU: Intel(R) Core (™) i5-3337U @ 1.80GHz
RAM: 4Gbs
SO: Linux Mint 19.1 Cinnamon

PARALELIZACIÓN DEL MÉTODO MONTE CARLO POR OpenMP.

Antes de comenzar con este método se debe tener en cuenta la instalación e importación de la librería OpenMp <<omp.h>>, sin más, haremos entonces las modificaciones pertinentes a nuestro programa secuencial:

```
int InCircle(const int &n, const auto &x, const auto &y)
{
    int j, cont = 0;
    for(j = 0; j < n; j++)
    {
        if (x*x + y*y < 1)
            cont++;
    }
    return cont;
}

void MontecarloP(int const &n, double &pi)
{
    std::random_device rd;
    std::mt19937 mt(rd());
    std::uniform_real_distribution<double> dist(0, 1);

    int PuntosCirculo = 0;
    int nChunk = n/100;
    int chunk = n/nChunk;
    int i;

    #pragma omp parallel for shared(nChunk, chunk) reduction(+:PuntosCirculo)
    for(i = 0; i <= nChunk; i++)
    {
        auto x = dist(mt);
        auto y = dist(mt);
        PuntosCirculo += InCircle(chunk, x, y);
    }
    pi = 4.0*(double(PuntosCirculo)/n);
}
```

Para realizar la paralelización con OpenMP intentamos hacer un uso eficiente de los hilos de forma que al ejecutarse en el ciclo, el orden de ejecución de los hilos no cause problemas en el resultado final, y para ello, lo mejor que podemos hacer es llevar nuestro código a un ciclo de sumas, dónde se guarda el resultado final en una sola variable a la que todos los hilos van ingresando tras calcular si el "lanzamiento" está o no dentro del círculo. La función “InCircle” se encarga de realizar esta parte. Se le envían entonces las cargas de trabajo a cada hilo, las

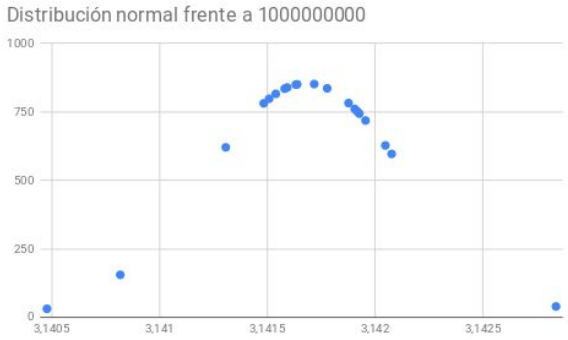
variables “chunk” y “nChunk” representan el cómo se ha repartido el trabajo total en diferentes pedazos para que cada hilo se encargue.

Entonces con esto podemos realizar una simple linea de OpenMP que nos llevará a agilizar el trabajo, “ #pragma omp parallel for” es un método propio de la librería que se encarga de trabajar con los ciclos “for” en el código, para el cual distribuye los hilos que se van a encargar de ejecutar la tarea en el ciclo, si le añadimos la sentencia “shared” la distribución se hace de forma que todos los hilos tienen acceso a la lista de variables, así es como pues “shared (nChunk, chunk)” asigna a cada hilo su carga de trabajo (por como está escrito el código) y la última sentencia de la línea, “reduction” se encarga de asignar a cada hilo una copia de una variable (private) sujeta a algún tipo de reducción al final de la sección paralela, para nuestro caso, “reduction(+:PuntosCirculo)”, se le da a cada hilo una copia de la variable “PuntosCirculo” que va a ser reducida al final en la forma de suma “+”.

```
main(int argc, char* argv[])
{
    int s;
    // cin >> s;
    s = stoi(argv[1]); //s = atoi(argv[1]);
    double pi;
    // Monte Carlo Paralelo
    clock_t StartingPoint = clock();
    MontecarloP(s, pi);
    StartingPoint = clock() - StartingPoint;


    printf("%f \n",pi);
    printf("%f \n",((float)StartingPoint)/CLOCKS_PER_SEC);

    return 0;
}
```

Como se aprecia solo añadimos unas cuantas líneas de código, veremos entonces si los resultados son tan favorables como esperamos.

Datos	1.000.000.000	Distribucion normal	Tiempo(seg)
1	3,141777	836,2633749	12,461082
2	3,141638	850,8637476	12,599673
3	3,141717	851,7428279	12,595124
4	3,142076	596,6851113	12,506454
5	3,141955	718,6559018	12,489837
6	3,142047	627,7140037	12,502636
7	3,142838	39,53245468	12,470728
8	3,141307	620,617505	12,484545
9	3,14158	834,8667157	12,620912
10	3,141926	743,9576445	12,500176
11	3,140818	155,2490833	12,436686
12	3,141905	760,9993138	12,532092
13	3,141632	849,8025654	12,545236
14	3,141508	798,1100074	12,538381
15	3,141539	816,1083637	12,416483
16	3,141592	839,2132259	12,468717
17	3,140478	31,03785195	12,437989
18	3,141483	781,3721344	12,571851
19	3,141876	782,5701831	12,785814
20	3,141915	753,0260172	12,577214
Media =	3,14168035	----------	12,5270815
Desviacion =	0,0004669429596	----------	0,08211830646
Tiempo de ejecucion en seg =			250,54163
Tiempo de ejecucion en min =			4,175693833



Observamos entonces que los resultados de este método modifican de manera sustancial los tiempos de ejecución pero los valores de pi siguen sin alterarse mucho, al igual que con el programa secuencial, hemos hallado 20 valores de pi con un número de iteraciones igual a 1000000000 por valor, vemos que el tiempo de ejecución total pasó de 64 min a solo 4 min, lo que significa un speedup de 15x.

Las especificaciones de la máquina donde fue ejecutado son las siguientes:

Máquina: Lenovo IdeaPad S400 Touch
CPU: Intel(R) Core (™) i5-3337U @ 1.80GHz
RAM: 4Gbs
SO: Linux Mint 19.1 Cinnamon

Aún tenemos una forma de paralelización más que aplicar a este método montecarlo.

PARALELIZACIÓN DEL MÉTODO MONTECARLO POR MPI

Antes de empezar con este método, al igual que con el método anterior necesitaremos agregar una librería : <<mpi.h>>, con ella tendremos manera de dividir el trabajo mas fácilmente. A continuación, veremos el código de la manera secuencias, y posteriormente su transformación a MPI

```
using std::mt19937 ;
using std::random_device;
using std::uniform_real_distribution;
int MontecarloS(int const &n)
{
    std::random_device rd;
    std::mt19937 mt(rd());
    std::uniform_real_distribution<double> dist(0, 1);
    int PuntosCirculo = 0;
    double x, y;
    for(int i = 0; i <= n; i++)
    {
        x = dist(mt);
        y = dist(mt);
        if (sqrt((x*x)+(y*y)) <= 1)
          PuntosCirculo++;
    }
    return PuntosCirculo;
}

void mpi_Montecarlo(int tries, double &result){
    // Initialize the MPI environment
    MPI_Init(NULL, NULL);
    // Find out rank, size
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
```

Se realizó una modificación leve al programa original de montecarlo secuencial, ahora en lugar de calcular pi, retorna la cantidad de lanzamientos exitosos. Se inicia el ambiente de MPI, las variables world_rank(rango del proceso) y world_size (cantidad de procesos lanzados) se de declaran y se obtienen a través de funciones propias de MPI.

```
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
int operaciones, number, sumador, totalShoots;
// We are assuming at least 2 processes for this task
if (world_size < 2) {
MPI_Abort(MPI_COMM_WORLD, 1);
}
//printf("Hello world from processor, rank %d out of %d processors\n",world_rank, world_size);
if (world_rank == 0) {
    // Proceso 0 asigna la cantidad de operaciones
    operaciones= tries/(world_size-1);
     clock_t StartingPoint = clock();
    for (int i=1; i<world_size;i++){ // i corresponde al proceso destino
        //en caso de que la division no sea exacta, el proceso 1 realiza las operaciones extra
        //cout<<'world size'<<world_size<<'world rank'<<world_rank<<endl;
        if (tries%(world_size-1)!= 0 && i == 1){
            int otro =operaciones+tries%(world_size-1); //operaciones + residuo entre intentos
            MPI_Send(
                /* data         = */ &otro,
                /* count        = */ 1,
                /* datatype     = */ MPI_INT,
                /* destination  = */ i,
                /* tag          = */ 0,
                /* communicator = */ MPI_COMM_WORLD);
        }
        else{
            MPI_Send(&operaciones, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
        }
```

la manera en la que se diseñó el modelo de concurrencia para este programa fue de tal manera que el proceso 0(head en el cluster) se encargará de dividir la cantidad de operaciones entre los procesos trabajadores (ejecutados en el cluster por los working nodes), para ello se divide la cantidad total de operaciones a realizar(disparos) entre la cantidad total de procesos - 1 (operaciones / world_size -1), para el caso en el no sea una división exacta las operaciones residuales las realizará el proceso 1; se procede entonces a realizar la rutina MPI_Send desde el proceso 0, enviando a cada proceso trabajador su respectiva carga laboral.

```
    MPI_Recv(
        /* data         = */ &totalShoots,
        /* count        = */ 1,
        /* datatype     = */ MPI_INT,
        /* source       = */ 1,
        /* tag          = */ 0,
        /* communicator = */ MPI_COMM_WORLD,
        /* status       = */ MPI_STATUS_IGNORE);
    StartingPoint = clock() - StartingPoint;
    result= 4.0*((double)totalShoots/(double)(tries));
    printf("pi: %lf \n",result);
    printf("%f.\n",((float)StartingPoint)/CLOCKS_PER_SEC);
} else if (world_rank != 0) {
    // procesos trabajadores reciben la cantidad de operaciones a realizar
    MPI_Recv(
        /* data         = */ &number,
        /* count        = */ 1,
        /* datatype     = */ MPI_INT,
        /* source       = */ 0,
        /* tag          = */ 0,
        /* communicator = */ MPI_COMM_WORLD,
        /* status       = */ MPI_STATUS_IGNORE);
    printf("Process %d received %d operations from process 0\n", world_rank, number);
    int shoots =MontecarloS(number);
    //printf("shoots: %d\n", shoots);
    if(world_rank == world_size-1){
        //el ultimo no recibe al colapsar y debe ser el primero en enviar
```

El proceso 0 se encarga también de recibir la cantidad total de disparos acertados del proceso 1(proceso explicado más adelante) y de calcular el tiempo gastado y el valor de pi; para cada uno de los procesos trabajadores se realiza una rutina MPI_Recv la cual recibe la

carga laboral proveniente del proceso 0 y procede a ejecutar la función secuencial de

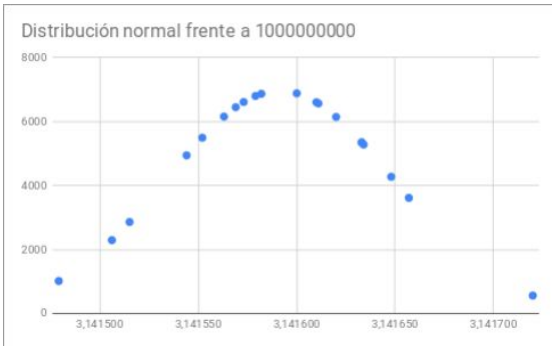
```
    //el ultimo no recibe al colapsar y debe ser el primero en enviar
    MPI_Send(
    /* data         = */ &shoots,
    /* count        = */ 1,
    /* datatype     = */ MPI_INT,
    /* destination  = */ world_rank-1,
    /* tag          = */ 0,
    /* communicator = */ MPI_COMM_WORLD);
}else
{
    //recibe el numero de cortes del proceso superior
    MPI_Recv(
    /* data         = */ &sumador,
    /* count        = */ 1,
    /* datatype     = */ MPI_INT,
    /* source       = */ world_rank+1,
    /* tag          = */ 0,
    /* communicator = */ MPI_COMM_WORLD,
    /* status       = */ MPI_STATUS_IGNORE);
    printf("process %d recieved %d + %d from %d\n",world_rank,sumador,shoots,world_rank+1);
    //adiciona cortes de este proceso al recibido del anterior superior
    sumador= sumador + shoots;
    //envia resultado al siguiente proceso inferior
    MPI_Send(
    /* data         = */ &sumador,
    /* count        = */ 1,
    /* datatype     = */ MPI_INT,
    /* destination  = */ world_rank-1,
    /* tag          = */ 0,
```

montecarlo descrita anteriormente
Cada uno de los procesos inicia una rutina recv la cual espera un mensaje proveniente de su proceso siguiente(world_rank +1), el contenido del mensaje corresponde a la sumatoria de disparos de todos los procesos mayores, por lo tanto es el último proceso (world_rank = world_size -1) el que debe comenzar enviando su resultado al proceso anterior, una vez recibido ese proceso suma el valor recibido con su propio resultado y realiza la rutina send al proceso anterior, esto se repite hasta que el proceso 1 envíe el resultado total al proceso 0.

```
        }
    }
    MPI_Finalize();
}
int main(int argc, char* argv[])
{
    double l;
    mpi_Montecarlo(stoi(argv[1]),l);
    return 0;
}
```

Una vez terminado el proceso se finaliza el ambiente de MPI, se utiliza un argumento del sistema para facilitar la toma de datos.

datos	1,000,000,000	Distribucion normal	Tiempo(seg)
1	3,141657	3615,405609	6,60274
2	3,141544	4944,789085	6,594525
3	3,141620	6146,470912	6,610716
4	3,141648	4274,450283	6,590872
5	3,141479	1016,991089	6,579471
6	3,141515	2862,524202	6,603886
7	3,141600	6883,703127	6,607278
8	3,141506	2293,291752	6,622114
9	3,141579	6800,57183	6,588114
10	3,141633	5349,103638	6,598665
11	3,141633	5349,103638	6,598665
12	3,141569	6449,62195	6,593876
13	3,141582	6868,627206	6,602987
14	3,141563	6157,148738	6,596292
15	3,141552	5496,222682	6,598778
16	3,141634	5280,965939	6,609558
17	3,141610	6604,445908	6,586455
18	3,141611	6566,144689	6,605678
19	3,141573	6611,891308	6,598653
20	3,141720	561,2490094	6,593722
Media =	3,141591	----------	6,59915225
Desviación =	),0000573056440	----------	0,00960975679
Tiempo de ejecución en seg =			131,983045
Tiempo de ejecución en min =			2.2min



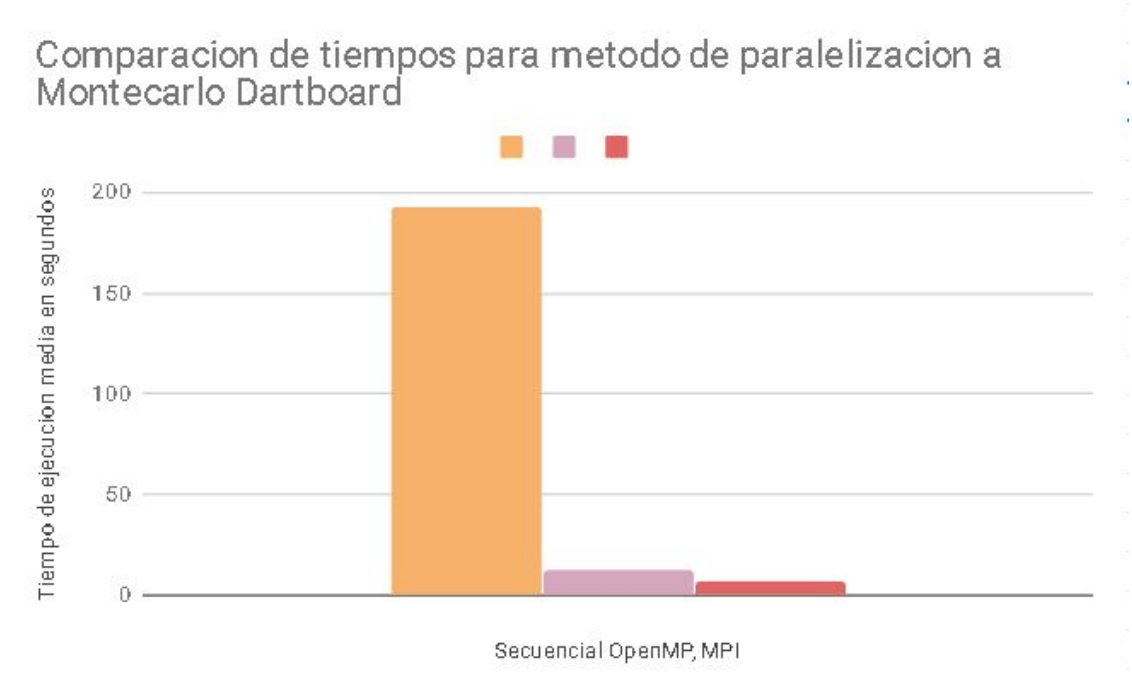
Como se puede apreciar, este método es incluso mejor que el paralelizado por OpenMP esto es, claro debido a la división de carga en máquinas diferentes, trabajando juntas para la realización del objetivo. pero se logró reducir el tiempo de 64 min a tan solo 2 min, algo realmente útil.

Las especificaciones del cluster utilizado se encuentran en la siguiente imagen, el programa se compiló utilizando el argumento -O3 para realizar la optimización en caché line se utilizó un nodo head y tres working nodes, se lanzan 8 procesos, el archivo machinefile no especifica la cantidad de procesos de cada nodo, por lo tanto se asigna según el algoritmo de despacho del cluster

```
cladmin@head:~$ sudo lshw
head
    description: Computer
    product: HVM domU
    vendor: Xen
    version: 4.2.amazon
    serial: ec2baf7e-5936-8ea4-cd89-0381008d4ac1
    width: 64 bits
    capabilities: smbios-2.7 dmi-2.7 vsyscall32
    configuration: boot=normal uuid=7EAF2BEC-3659-A48E-CD89-0381008D4AC1
  *-core
       description: Motherboard
       physical id: 0
     *-firmware
          description: BIOS
          vendor: Xen
          physical id: 0
          version: 4.2.amazon
          date: 08/24/2006
          size: 96KiB
          capabilities: pci edd
     *-cpu
          description: CPU
          product: Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40GHz
          vendor: Intel Corp.
          physical id: 401
          bus info: cpu@0
          slot: CPU 1
          size: 2400MHz
          capacity: 2400MHz
          width: 64 bits
          capabilities: fpu fpu_exception wp vme de pse tsc msr pae mce cx8 apic sep mtrr
onstant_tsc rep_good nopl xtopology cpuid pni pclmulqdq ssse3 fma cx16 pcid sse4_1 sse4_2
hf_lm abm cpuid_fault invpcid_single pti fsgsbase bmi1 avx2 smep bmi2 erms invpcid xsaveop
     *-memory
          description: System Memory
          physical id: 1000
          size: 1GiB
          capacity: 1GiB
          capabilities: ecc
          configuration: errordetection=multi-bit-ecc
        *-bank
             description: DIMM RAM
             physical id: 0
             slot: DIMM 0
```

COMPARACIÓN DE RESULTADOS

Secuencial	OpenMP	MPI	
192,298935	12,461082	6,60274	
192,320236	12,599673	6,594525	
192,603668	12,595124	6,610716	
194,688339	12,506454	6,590872	
191,72496	12,489837	6,579471	
192,514755	12,502636	6,603886	
192,576385	12,470728	6,607278	
194,118668	12,484545	6,622114	
192,983841	12,620912	6,588114	
192,910278	12,500176	6,598665	
191,607574	12,436686	6,598665	
193,317078	12,532092	6,593876	
192,026459	12,545236	6,602987	
192,824799	12,538381	6,596292	
192,239532	12,416483	6,598778	
194,855255	12,468717	6,609558	
195,51712	12,437989	6,586455	
193,40654	12,571851	6,605678	
192,634079	12,785814	6,598653	
192,13063	12,577214	6,593722	
192,9649566	12,5270815	6,59915225	<<<-- Promedio
Tiempo de ejecución en segundos			



En el caso del secuencial al hacer lanzamientos de 1x10^9 vemos que se demora mucho en comparación con las demás formas del código, esto se debe por la generación de los números aleatorios y genera el “random device” para cada hilo, y este es el principal problema del código secuencial. Para el código de OpenMP, se paralelizó lo que ya tenemos y se logra arreglar el problema del “random device” creando un “random device” compartido esto mejora mucho los tiempos de ejecución con la misma cantidad de tiros. Y en MPI aparte de ser ejecutado en el cluster montado en aws, realiza la división de carga de trabajo en cada proceso optimizando el tiempo y el uso del cluster.
De las tres maneras en las que se hizo el mismo programa el que arroja mejores tiempos es el de montecarlo hecho con MPI.

Se obtuvo un speedup de 15x con el método OpenMP
Se obtuvo un speedup de 29x con el método MPI

MÉTODO AGUJA DE BUFFON

Georges Louis Leclerc(1707-88), Conde de Buffon fue un celebre naturalista francés autor de una monumental Historia Natural en 44 tomos que recopilaba el conocimiento científico con un fin eminentemente divulgativo. Hoy en día su nombre aparece muchas veces asociado a un problema denominado "La aguja de Buffon" que relaciona el número pi con el lanzamiento de una aguja sobre una superficie rayada.

¿Qué es el Método de la aguja de Buffon?

Es un método que permite obtener un valor aproximado del número π, aunque no fue la motivación original de Buffon. Si L en la longitud de las agujas y c la distancia entre las líneas, la probabilidad de que la aguja se cruce con una línea es:

$$P = \frac{2L}{\pi c}$$

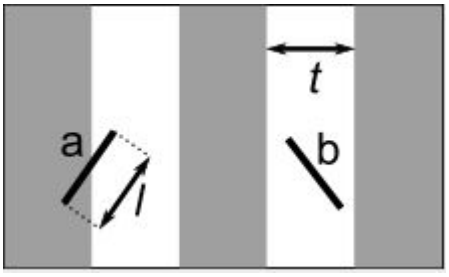
Por tanto,

$$\pi = \frac{2L}{Pc}$$

La aguja de Buffon fue uno de los primeros problemas de geometría y probabilidad, y con geometría integral se puede resolver. La solución: si la longitud de la aguja no es mayor que la anchura de las líneas paralelas, puede aproximarse al número π por el método de Monte Carlo. La probabilidad en este caso sería una aproximación experimental del caso teórico, sería una estadística. Y cuantas más agujas se cuentan, más nos aproximamos al número π.

$$P \approx \frac{|\text{agujas cruzadas}|}{|\text{agujas totales}|}$$

Hallando Pi con el el método de agujas de Buffon



Tomemos una aguja de longitud l, en un cuadro se dibujan líneas verticales de longitud l entre ellas. Después arrojamos la aguja cierto número de veces. En cada lanzamiento la aguja puede cortar una de las líneas. Contemos el número de lanzamientos, N, y el número de veces en las que la aguja corta a alguna de las rectas, A. Multiplicamos N por 2 y dividimos el resultado entre A. Veréis que el resultado es muy cercano a π y será tanto más cercano a él cuanto mayor sea el número de lanzamientos.

Si la aguja es más corta que la distancia entre las líneas, llamémosla D, se puede demostrar que:

$$\pi = \frac{2 \cdot N \cdot L}{A \cdot D}$$

Una vez hallado esto, podemos diseñar el algoritmo que haga esto por nosotros, así pues, a continuación mostramos como lo hemos hecho y qué resultados obtuvimos con el.

Primero necesitaremos agregar las mismas librerías que con el método anterior.,estas son: <<iostream>><<<math.h>><<<time.h>><<<stdlib.h >><<<random>>

```
void BuffonS(const int &stop, const double &MePi, double &Npi)
{
    std::random_device rd;
    std::mt19937 mt(rd());
    std::uniform_real_distribution<double> dist(0.0, 1.0);

    int NumLanzados, NumCortados, i;
    double LongAguja, DistLinea, AgujaBajo, AgujaAlto, Angulo;
    NumLanzados=0;
    NumCortados=0;
    LongAguja=2.0;
    DistLinea=2.0;

    for (NumLanzados = 0; NumLanzados < stop ; NumLanzados++)
    {
        AgujaBajo=dist(mt)*DistLinea;
        Angulo=dist(mt)*180;
        AgujaAlto=AgujaBajo + sin(MePi*Angulo/180)*LongAguja;//M_PI de math.h
        //cout<<Angulo<<endl;
        if (AgujaAlto >= DistLinea)
        {
            NumCortados++;
        }

    }
    //cout<<"tries:"<<NumLanzados<<" cuts:"<<NumCortados;
    Npi= (double)NumLanzados*2/(double)NumCortados;
}
```

Antes que nada necesitamos definir las variables que utilizaremos en la elaboración del programa, segun el analisis matematico, necesitamos entonces 4 variables : LongAguja (medida de la aguja), DistLinea (Distancia entre una línea y otra), NumLAnzados (número total de iteraciones), NumCortados (Número de agujas que caen sobre la línea.

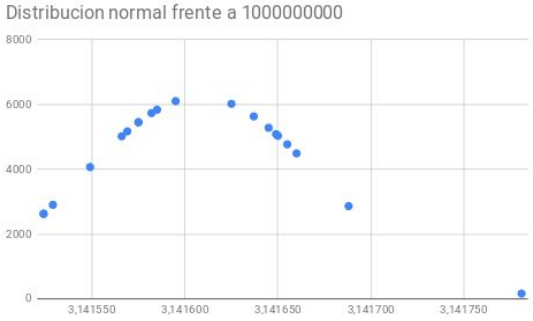
Nos fijamos que lo que realmente aumenta la probabilidad de que estas agujas crucen o no la línea es su ángulo, ya que este determina la posición en la que la aguja queda al caer, entonces

diseñamos la función de manera que esta variable Ángulo sea la que obtenga valores aleatorios,
Haciendo un par de cálculos más, con ayuda de pitágoras, encontraremos la distancia a la que la cayó la aguja de las líneas.
Con esto a nuestra disposición ya es posible asegurar con mayor confianza que agujas cortaron la linea, y las sumaremos a la variable que lleva esta información (NumeroCortados).

```
int main(int argc, char* argv[])
{
    int stop;
    // cin >> stop;
    stop = stoi(argv[1]);

    double  Npi, MePi = 3.14159265359;

    //Calculo de pi = 2N/N'
    clock_t StartingPoint = clock();
    BuffonS(stop, MePi, Npi);
    StartingPoint = clock() - StartingPoint;

    printf("\n%f \n", Npi);
    printf("%f \n",((float)StartingPoint)/CLOCKS_PER_SEC);

    return 0;
}
```

Ya con la función creada solo nos resta llamarla desde la función main(). agregando donde empezará a correr el tiempo, y donde se detendrá sus respectivos prints para saber estos valores y es todo, vamos a hacerle un test a ver que tal funciona a continuación:

Datos	1,000,000,000	Distribucion normal	Tiempo(seg)
1	3,141660	4489,248019	236,652054
2	3,141566	5017,553108	235,478668
3	3,141524	2626,570296	235,287521
4	3,141575	5450,10487	237,144943
5	3,141549	4066,623254	235,515076
6	3,141529	2901,73161	234,921631
7	3,141595	6102,049521	235,344635
8	3,141781	162,6978745	240,79689
9	3,141645	5280,558442	243,038055
10	3,141650	5033,005255	242,751602
11	3,141524	2626,570296	235,436905
12	3,141575	5450,10487	234,893173
13	3,141688	2862,590316	234,535095
14	3,141649	5084,050397	235,134521
15	3,141655	4767,879812	235,979874
16	3,141585	5837,669763	234,974442
17	3,141637	5630,31387	235,171722
18	3,141625	6019,959682	236,002274
19	3,141569	5169,122206	234,846817
20	3,141582	5733,264291	237,967911
Media =	3,14160815	----------	236,5936905
Desviación =	),0000640133826	----------	2,583347707
Tiempo de ejecución en seg =			4731,873809
Tiempo de ejecución en min =			78,86456348



Como se puede observar en la tabla de datos, no es un método muy veloz, ya que tarda incluso más que su contraparte dartboard, sin embargo no podemos obviar el hecho de que su precisión es alta, se debe tener en cuenta que los datos al ser aleatorios pueden variar bastante el resultado, incluso teniendo una cantidad tan grande de iteraciones (1000000000) Al igual que con el método anterior realizamos implementaciones de paralelización a este programa esperando una mejora sustancial en sus tiempos de ejecución.

Las especificaciones de la máquina donde fue ejecutado son las siguientes:

Máquina: Lenovo IdeaPad S400 Touch
CPU: Intel(R) Core (™) i5-3337U @ 1.80GHz
RAM: 4Gbs
SO: Linux Mint 19.1 Cinnamon

PARALELIZACIÓN POR OpenMP

Tenemos que tener en cuenta lo mismo que hicimos para el método de Montecarlo, instalando la librería <<omp.h>> y manejando la simplificación a sumas.

```
int InLine(const int &n, const double &MePi)
{
    std::random_device rd;
    std::mt19937 mt(rd());
    std::uniform_real_distribution<double> dist(0.0, 1.0);

    int j, cont = 0;

    double LongAguja, DistLinea, AgujaBajo, AgujaAlto, Angulo;

    LongAguja=2.0;
    DistLinea=2.0;

    for (j = 0; j < n; j++)
    {
        AgujaBajo = dist(mt)*DistLinea;
        Angulo = dist(mt)*180;
        AgujaAlto = AgujaBajo + sin(MePi*Angulo/180)*LongAguja;
        if (AgujaAlto >= DistLinea)
        {
            cont++;
        }
    }
    return cont;

}
```

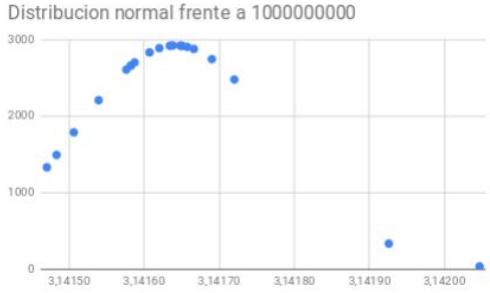
Tenemos nuestra función que se encarga de hacer los cálculos para las lineas y que cada hilo ejecuta para hallar pi.
Como en el método anterior, cada hilo tiene una carga de trabajo asignada con la que entrará a la función “InLine” e irá aumentando el contador de las veces que la “aguja” cortó la “linea” y este contador será lo que en este caso queremos reducir para cada hilo.

```
void BuffonP(const int &stop, const double &MePi, double &Npi)
{

    int NumLanzados, NumCortados;
    NumCortados = 0;

    int nChunk = stop/1000;
    //cout << "nChunk: " << nChunk << endl;
    int chunk = stop/nChunk;
    //cout << "Chunk: " << chunk << endl;

    #pragma omp parallel for shared(nChunk, chunk) reduction(+:NumCortados)
    for (NumLanzados = 0; NumLanzados < stop ; NumLanzados++)
    {
        NumCortados += InLine(chunk, MePi);
    }
    //cout<<"tries:"<<NumLanzados<<" cuts:"<<NumCortados;
    Npi= (double)NumLanzados*2/(double)NumCortados;
}
```

Usamos el mismo método de paralelización del caso anterior ya que podemos trabajarlo de una forma muy similar, compartiendo las cargas de trabajo de cada hilo y haciendo una reducción de suma en la variable “NumCortados” con la que al final aproximamos el cálculo a pi.

```
int main(int argc, char* argv[])
{
    int stop;
    // cin >> stop;
    stop = stoi(argv[1]);

    double  Npi, MePi = 3.14159265359;

    //Calculo de pi = 2N/N'
    clock_t StartingPoint = clock();
    BuffonP(stop, MePi, Npi);
    StartingPoint = clock() - StartingPoint;

    printf("\n%f \n", Npi);
    printf("%f \n",((float)StartingPoint)/CLOCKS_PER_SEC);

    return 0;
}
```

Datos	1.000.000.000	Distribucion norma	Tiempo(seg)
1	3,14162	2885,803166	23,466686
2	3,1417	2742,173566	23,445686
3	3,1415	1328,917691	24,465645
4	3,141582	2657,739461	23,456415
5	3,141657	2902,456283	23,300124
6	3,141648	2918,047285	23,204712
7	3,141506	1786,952277	23,597228
8	3,141582	2657,739461	23,412532
9	3,141634	2917,171103	23,319826
10	3,141720	2475,473322	23,391286
11	3,141638	2920,552111	23,428728
12	3,142047	35,47251351	23,465686
13	3,141926	332,9668718	23,396758
14	3,141539	2205,436871	23,581541
15	3,141483	1490,816352	23,465874
16	3,141576	2604,910177	23,412441
17	3,141650	2915,669689	23,465312
18	3,141607	2830,215743	23,494714
19	3,141666	2874,435619	23,413141
20	3,141587	2698,598755	24,000046
Media =	3,14164	----------	23,50921905
Desviación =	0,0001365559072	----------	0,273041209
Tiempo de ejecución en seg =			470,184381
Tiempo de ejecución en min =			7,83640635



Encontramos en estos resultados lo esperado, una minimización de tiempo bastante visible en comparación a la forma secuencial con la misma cantidad de datos (1000000000) logrando reducir su tiempo de ejecución total de 78 min a tan solo 7 min. sin embargo al igual que con el método montecarlo anterior implementaremos una paralelización más al Buffon.

Las especificaciones de la máquina donde fue ejecutado son las siguientes:

Máquina: Lenovo IdeaPad S400 Touch
CPU: Intel(R) Core (™) i5-3337U @ 1.80GHz
RAM: 4Gbs
SO: Linux Mint 19.1 Cinnamon

PARALELIZACIÓN POR MPI

```
int Seq_Buffon(int tries, double LongAguja, double DistLinea ){
    std::random_device rd;
    std::mt19937 mt(rd());
    std::uniform_real_distribution<double> dist(0.0, 1.0);

    int NumLanzados, NumCortados;
    double AgujaBajo, AgujaAlto, Npi, Angulo;
    int r[1];
    NumLanzados=0;
    NumCortados=0;
    while (NumLanzados < tries){
        AgujaBajo=dist(mt)*DistLinea;
        Angulo=dist(mt)*180;
        AgujaAlto=AgujaBajo + sin(M_PI*Angulo/180)*LongAguja;//M_PI de math.h
        NumLanzados++;
        if (AgujaAlto >= DistLinea){
            NumCortados++;
        }
    }
    return NumCortados;
}
void mpi_Buffon(int tries, double LongAguja, double DistLinea){
    // Initialize the MPI environment
    MPI_Init(NULL, NULL);
    // Find out rank, size
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    int world_size;
```

Se realizó una modificación leve al programa original de buffon secuencial, ahora en lugar de calcular pi, retorna la cantidad de cortes. Se inicia el ambiente de MPI, las variables world_rank(rango del proceso) y world_size (cantidad de procesos lanzados) se de declaran y se obtienen a través de funciones propias de MPI.

```
int world_size;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
int operaciones,number, sumador,totalCuts;
// We are assuming at least 2 processes for this task
if (world_size < 2) {
MPI_Abort(MPI_COMM_WORLD, 1);
}
//printf("Hello world from processor, rank %d out of %d processors\n",world_rank, world_size);
if (world_rank == 0) {
    // Proceso 0 asigna la cantidad de operaciones
    operaciones= tries/(world_size-1);
     clock_t StartingPoint = clock();
    for (int i=1; i<world_size;i++){ // i corresponde al proceso destino
        //en caso de que la division no sea exacta, el proceso 1 realiza las operaciones extra
        //cout<<'world size'<<world_size<<'world rank'<<world_rank<<endl;
        if (tries%(world_size-1)!= 0 && i == 1){
            int otro =operaciones+tries%(world_size-1); //operaciones + residuo entre intentos
            MPI_Send(
                /* data         = */ &otro,
                /* count        = */ 1,
                /* datatype     = */ MPI_INT,
                /* destination  = */ i,
                /* tag          = */ 0,
                /* communicator = */ MPI_COMM_WORLD);
        }
        else{
            MPI_Send(
                /* data         = */ &operaciones,
                /* count        = */ 1,
```

la manera en la que se diseñó el modelo de concurrencia para este programa fue de tal manera que el proceso 0(head en el cluster) se encargará de dividir la cantidad de operaciones entre los procesos trabajadores (ejecutados en el cluster por los working nodes), para ello se divide la cantidad total de operaciones a realizar(lanzamientos de aguja) entre la cantidad total de procesos - 1 (operaciones / world_size -1), para el caso en el no sea una división exacta las operaciones residuales las realizará el proceso 1; se procede entonces a realizar la rutina MPI_Send desde el proceso 0, enviando a cada proceso trabajador su respectiva carga laboral.

```
                /* count        = */ 1,
                /* datatype     = */ MPI_INT,
                /* destination  = */ i,
                /* tag          = */ 0,
                /* communicator = */ MPI_COMM_WORLD);
        }
    }
    //termina asignacion de operaciones
    //recibe el numero de cortes, colapsado de todos los procesos trabajadores
    MPI_Recv(
        /* data         = */ &totalCuts,
        /* count        = */ 1,
        /* datatype     = */ MPI_INT,
        /* source       = */ 1,
        /* tag          = */ 0,
        /* communicator = */ MPI_COMM_WORLD,
        /* status       = */ MPI_STATUS_IGNORE);
    StartingPoint = clock() - StartingPoint;
    printf("%f.\n",((float)StartingPoint)/CLOCKS_PER_SEC);
    cout<<"2*n/n' = 2*"<<tries<<"/"<<totalCuts<<"="<<(double)(2.0*tries/totalCuts)<<endl;
} else if (world_rank != 0) {
    // procesos trabajadores reciben la cantidad de operaciones a realizar
    MPI_Recv(
        /* data         = */ &number,
        /* count        = */ 1,
        /* datatype     = */ MPI_INT,
        /* source       = */ 0,
        /* tag          = */ 0,
        /* communicator = */ MPI_COMM_WORLD,
```

El proceso 0 se encarga también de recibir la cantidad total de cortes del proceso 1(proceso explicado más adelante) y de calcular el tiempo gastado y el valor de pi; para cada uno de los procesos trabajadores se realiza una rutina MPI_Recv la cual recibe la carga laboral proveniente del proceso 0 y procede a ejecutar la función secuencial de Buffon descrita anteriormente

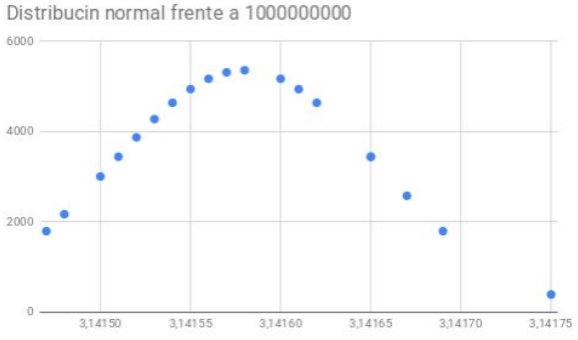
```
    /* status       = */ MPI_STATUS_IGNORE);
//printf("Process %d received %d operations from process 0\n", world_rank, number);
int cuts =Seq_Buffon(number,2.0,2.0);
//printf("cuts: %d\n", cuts);
if(world_rank == world_size-1){
    //el ultimo no recibe al colapsar y debe ser el primero en enviar
    MPI_Send(
    /* data         = */ &cuts,
    /* count        = */ 1,
    /* datatype     = */ MPI_INT,
    /* destination  = */ world_rank-1,
    /* tag          = */ 0,
    /* communicator = */ MPI_COMM_WORLD);
}else
{
    //recibe el numero de cortes del proceso superior
    MPI_Recv(
    /* data         = */ &sumador,
    /* count        = */ 1,
    /* datatype     = */ MPI_INT,
    /* source       = */ world_rank+1,
    /* tag          = */ 0,
    /* communicator = */ MPI_COMM_WORLD,
    /* status       = */ MPI_STATUS_IGNORE);
    printf("process %d recieved %d + %d from %d\n",world_rank,sumador,cuts,world_rank+1);
    //adiciona cortes de este proceso al recibido del anterior superior
    sumador= sumador + cuts;
    //envia resultado al siguiente proceso inferior
    MPI_Send(
```

Cada uno de los procesos inicia una rutina recv la cual espera un mensaje proveniente de su proceso siguiente(world_rank +1), el contenido del mensaje corresponde a la sumatoria de cortes de todos los procesos mayores, por lo tanto es el último proceso (world_rank = world_size -1) el que debe comenzar enviando su resultado al proceso anterior, una vez recibido ese proceso suma el valor recibido con su propio resultado y realiza la rutina send al proceso anterior, esto se repite hasta que el proceso 1 envíe el resultado total al proceso 0.

```
            MPI_Send(
            /* data         = */ &sumador,
            /* count        = */ 1,
            /* datatype     = */ MPI_INT,
            /* destination  = */ world_rank-1,
            /* tag          = */ 0,
            /* communicator = */ MPI_COMM_WORLD);
        }
    }
    MPI_Finalize();
}
int main(int argc, char* argv[]){
  mpi_Buffon(stoi(argv[1]), 2.0, 2.0);
  return 0;
}
```

Una vez terminado el proceso se finaliza el ambiente de MPI, se utiliza un argumento del sistema para facilitar la toma de datos.

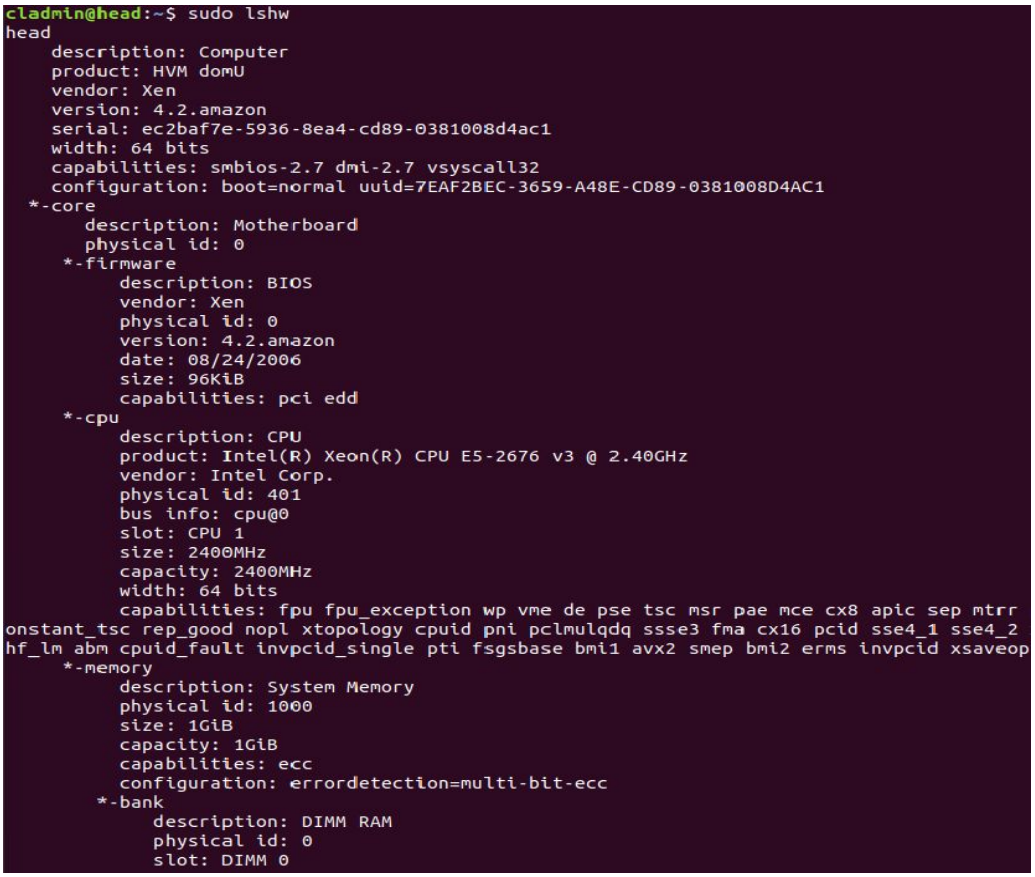
datos	1,000,000,000	Distribucion normal	Tiempo(seg)
1	3,14153	4277,9182	12,300316
2	3,14162	4640,120119	12,254155
3	3,14167	2579,932066	12,265739
4	3,14155	4942,904834	12,347045
5	3,14165	3444,354285	12,289643
6	3,14154	4640,120119	13,547902
7	3,14151	3444,354285	12,310965
8	3,14160	5171,2026	12,325249
9	3,14152	3873,396957	12,324627
10	3,14169	1797,768794	12,315574
11	3,14175	394,3312729	12,310459
12	3,14161	4942,904834	12,303876
13	3,14156	5171,2026	12,277534
14	3,14150	3008,014353	12,293192
15	3,14147	1797,768794	12,289332
16	3,14158	5361,409708	12,273601
17	3,14157	5313,211887	12,312168
18	3,14165	3444,354285	12,284318
19	3,14148	2173,166066	12,312872
20	3,14155	2173,166066	12,323134
Media =	3,14158	----------	12,36308505
Desviación =	,0000744099597:	----------	0,2797977602
Tiempo de ejeción en seg =			247,261701
Tiempo de ejeción en min =			4min



Una vez más la paralelización MPI ha demostrado que es posible reducir incluso más el tiempo de los datos recolectados,

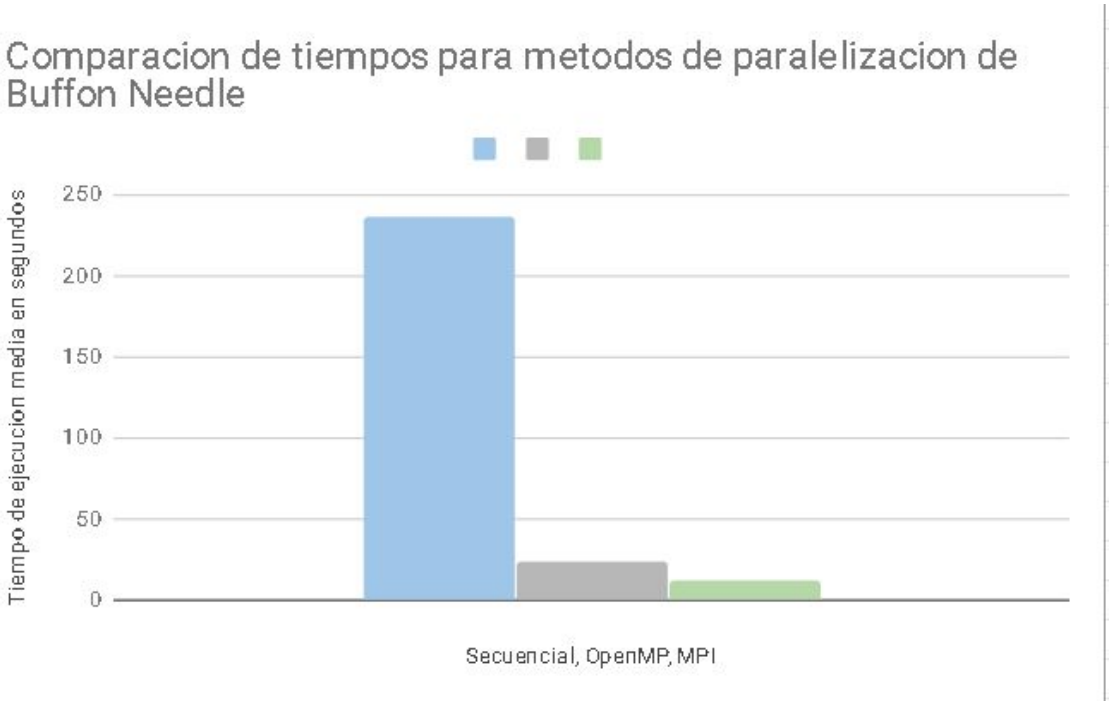
reduciendo su tiempo de ejecución total de 78 min a solo 4, con una gran precisión,

Las especificaciones del cluster utilizado se encuentran en la siguiente imagen, el programa se compiló utilizando el argumento -O3 para realizar la optimización en caché line se utilizó un nodo head y tres working nodes, se lanzan 8 procesos, el archivo machinefile no especifica la cantidad de procesos de cada nodo, por lo tanto se asigna según el algoritmo de despacho del cluster.



COMPARACIÓN DE RESULTADOS

Secuencial	OpenMP	MPI	
236,652054	23,466686	12,300316	
235,478668	23,445686	12,254155	
235,287521	24,465645	12,265739	
237,144943	23,456415	12,347045	
235,515076	23,300124	12,289643	
234,921631	23,204712	13,547902	
235,344635	23,597228	12,310965	
240,79689	23,412532	12,325249	
243,038055	23,319826	12,324627	
242,751602	23,391286	12,315574	
235,436905	23,428728	12,310459	
234,893173	23,465686	12,303876	
234,535095	23,396758	12,277534	
235,134521	23,581541	12,293192	
235,979874	23,465874	12,289332	
234,974442	23,412441	12,273601	
235,171722	23,465312	12,312168	
236,002274	23,494714	12,284318	
234,846817	23,413141	12,312872	
237,967911	24,000046	12,323134	
236,5936905	23,50921905	12,36308505	<<--Promedio
Tiempo de ejecucion en segundos			



En agujas de Buffon observamos unos gráficos muy parecidos a los vistos en montecarlo, en donde el secuencial se demora mucho debido a la forma en que se tratan los datos de las agujas ya que cada una de estas era procesada de manera individual y OJO solo lo hicimos cuando la longitud de la aguja es menor al espacio entre líneas para facilitar su desarrollo.
Cuando realizamos la paralelización mediante OpenMP hacemos que los hilos tengan acceso a variables necesarias para el cálculo de pi sin modificarla, comparado con secuencial hubo una gran ganancia en tiempo comprobando así la efectividad de la paralelización.
Y por último en la ejecución realizada con MPI vemos que el distribución de procesos sigue siendo la mejor opción de paralelismo y concurrencia, además de usar los beneficios del cluster para ejecutarlo, en este caso vemos que es mejor paralelizado ya que la comunicación por procesos es más sencilla y llevan los datos que el proceso requiere para su debida ejecución.

Se obtuvo un speedup de 10x con el metodo OpenMP
se obtuvo un speedup de 19x con el metodo MPI

CONCLUSIÓN

El mejor método es MPI, como podemos ver en los resultados observados los tiempos que arrojan los programas son mucho menores comparados con el secuencial y el OpenMP esto creemos, se debe a la distribución de la carga de trabajo entre los nodos trabajadores lo que hace aprovechar todos los recursos que las máquinas del cluster nos puede ofrecer, la gran ventaja de MPI es que nos ofrece una manera efectiva de comunicar los datos entre procesos, aunque tuvimos que tener mucho cuidado entre los procesos para que no utilicen datos cambiantes, esto creemos que pudo afectar un poco el tiempo.

A diferencia de MPI, OpenMP lo ejecutamos en una máquina de manera local lo cual le hace perder algo de poder de ejecución cuando lo comparamos con el cluster en el que se ejecutó MPI, además del problema que se tuvo al momento de la creación de los hilos ya que cada hilo creaba su propio “random device” y esto hace que el proceso sea un poco más lento de lo normal, al darnos cuenta de este error, buscamos solucionarlo creando “random device” compartido en el que cada hilo tuviera accesos a este, pero esto igualmente afectó de manera negativa la sincronización final de los hilos generando que sea un poquito más lento que las ejecuciones hechas con MPI.