ASSB – Practica 1 – TRABAJO EN GRUPO 4

1. INTRODUCCIÓN

1.1 Selección de la función de número aleatorio.

Para diseñar nuestro programa de cálculo de PI usando el método MonteCarlo necesitaremos una función reentrante que genere números aleatorios entre 0 y 1.

Para ello utilizamos el comando $ man -k rand, con el que obtenemos un listado de funciones de diversas características que generan números aleatorios.

```
vallealonsodecaso@Zenbook:~$ man -k rand
RAND (7ssl)          - the OpenSSL random generator
drand48 (3)          - generate uniformly distributed pseudo-random numbers
drand48_r (3)        - generate uniformly distributed pseudo-random numbers reentrantly
erand48 (3)          - generate uniformly distributed pseudo-random numbers
erand48_r (3)        - generate uniformly distributed pseudo-random numbers reentrantly
EVP_RAND (7ssl)      - the random bit generator
EVP_RAND-CTR-DRBG (7ssl) - The CTR DRBG EVP_RAND implementation
EVP_RAND-HASH-DRBG (7ssl) - The HASH DRBG EVP_RAND implementation
EVP_RAND-HMAC-DRBG (7ssl) - The HMAC DRBG EVP_RAND implementation
EVP_RAND-SEED-SRC (7ssl) - The randomness seed source EVP_RAND implementation
EVP_RAND-TEST-RAND (7ssl) - The test EVP_RAND implementation
getentropy (3)       - fill a buffer with random bytes
getrandom (2)        - obtain a series of random bytes
initstate (3)        - random number generator
initstate_r (3)      - reentrant random number generator
jrand48 (3)          - generate uniformly distributed pseudo-random numbers
jrand48_r (3)        - generate uniformly distributed pseudo-random numbers reentrantly
lcong48 (3)          - generate uniformly distributed pseudo-random numbers
lcong48_r (3)        - generate uniformly distributed pseudo-random numbers reentrantly
life_cycle-rand (7ssl) - The RAND algorithm life-cycle
lrand48 (3)          - generate uniformly distributed pseudo-random numbers
lrand48_r (3)        - generate uniformly distributed pseudo-random numbers reentrantly
mrand48 (3)          - generate uniformly distributed pseudo-random numbers
mrand48_r (3)        - generate uniformly distributed pseudo-random numbers reentrantly
nrand48 (3)          - generate uniformly distributed pseudo-random numbers
nrand48_r (3)        - generate uniformly distributed pseudo-random numbers reentrantly
openssl-rand (1ssl)  - generate pseudo-random bytes
```

De entre todas las funciones que devuelve el comando, debemos seleccionar aquellas que cumplen las características necesarias para nuestro programa.

En concreto, hemos seleccionado erand48 y mediante el comando $ mand 3 erand48 obtenemos todas las características concretas de la función, de las cuales nos interesan especialmente las siguientes:

```
vallealonsodecaso@Zenbook: $ man 3 erand48
```

```
DESCRIPTION
       These functions generate pseudo-random numbers using the linear congruential algorithm and 48-bit  in-
       teger arithmetic.

       The  drand48()  and erand48() functions return nonnegative double-precision floating-point values uni-
       formly distributed over the interval [0.0, 1.0).
```

```
       double erand48(unsigned short xsubi[3]);
```

Como cumple las características que necesitábamos será la función que usemos para generar los números aleatorios del programa.

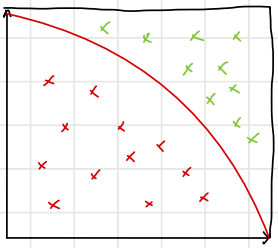
2. DISEÑO DEL PROGRAMA PARA EL CÁLCULO DE PI

Para calcular pi según las condiciones expuestas en el enunciado, en primer lugar, hemos desarrollado la idea del algoritmo que queríamos implementar a mano según las siguientes notas:

PLANTEAMIENTO DEL EJERCICIO

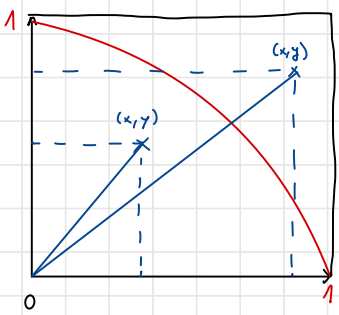
Monte Carlo

¿ π ?



$$\pi = 4 \times \frac{n^{\circ} \text{ dardos dentro}}{n^{\circ} \text{ dardos total}}$$

para saber si un dardo está dentro o no:



$$h = \sqrt{x^2 + y^2}$$

si $\begin{cases} h < 1 & \text{dardo dentro} \\ h > 1 & \text{dardo fuera} \end{cases}$

ESQUEMA DE IMPLEMENTACIÓN

Serie

$n = n^{\circ}$ total dardos

Paralelo

def func (n)

def func (n, h)

$h = n^{\circ}$ hilos

Esta idea desarrollada “a papel” se implementa en código C en los siguientes programas:

a) Programa del cálculo de Pi en serie:

```
  [1/1]
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(int argc, char *argv[]) {
    int n, i, h, aux, a, b, c;
    double ac = 0, x, y, dentro=0;
    n = atoi(argv[1]);

    for (i = 0; i < n; i++) {
        a = i * 5;
        b = i * 7;
        c = i * 9;
        unsigned short seed[3] = {a, b, c};

        x = erand48(seed);
        y = erand48(seed);
        aux = x * x + y * y;
        h = sqrt(aux);

        if (h < 1) {
            dentro += 1;
        }
    }

    double pi = 4.0* (dentro/n);
    printf("El cálculo da %f\n", pi);
```

Que devuelve:

```
calculoPI.c
aleingmar@DESKTOP-2IE63G4:~/ASSB/Practica_1/practical_proyectoGrupal$ gcc -g -O0 -o calculoPI calculoPI.c -lm
aleingmar@DESKTOP-2IE63G4:~/ASSB/Practica_1/practical_proyectoGrupal$ ./calculoPI 10
El cálculo da 2.800000
aleingmar@DESKTOP-2IE63G4:~/ASSB/Practica_1/practical_proyectoGrupal$ ./calculoPI 10000
El cálculo da 3.159200
aleingmar@DESKTOP-2IE63G4:~/ASSB/Practica_1/practical_proyectoGrupal$ ./calculoPI 100000000
El cálculo da 3.141358
aleingmar@DESKTOP-2IE63G4:~/ASSB/Practica_1/practical_proyectoGrupal$ nano cal*
aleingmar@DESKTOP-2IE63G4:~/ASSB/Practica_1/practical_proyectoGrupal$
```

b) Programa del cálculo de Pi en paralelo:

```
  GNU nano 4.8                                          calculoPI_par_prueba.c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

int main(int argc, char *argv[]) {
    int n;
    double dentro = 0.0;
    n = atoi(argv[1]);
    omp_set_num_threads(atoi(argv[2]));
 int i, h, aux, a, b, c, id;
        double x, y;
    #pragma omp parallel private(i, h, aux, a, b, c, x, y)
    {
        id = omp_get_thread_num();
        unsigned short seed[3] = {id, id + 1, id + 2}; // Utilizar id como semilla

        #pragma omp for reduction(+:dentro)
        for (i = 0; i < n; i++) {
            x = erand48(seed);
            y = erand48(seed);
            aux = x * x + y * y;
            h = sqrt(aux);
            if (h < 1) {
```

```
            if (h < 1) {
                dentro += 1;
            }
        }
    }

    double pi = 4.0 * (dentro / (double)n);
    printf("El cálculo da %f\n", pi);

    return 0;
}
```

Que devuelve:

```
aleingmar@DESKTOP-2IE63G4:~/ASSB/Practica_1/practical_proyectoGrupal$ nano calculoPI_par_prueba.c
aleingmar@DESKTOP-2IE63G4:~/ASSB/Practica_1/practical_proyectoGrupal$ gcc -g -O0 -fopenmp -o calculoPI_par_prueba calculoPI_par_prueb
a.c -lm
aleingmar@DESKTOP-2IE63G4:~/ASSB/Practica_1/practical_proyectoGrupal$ ./calculoPI_par_prueba 100000 12
El cálculo da 3.141840
aleingmar@DESKTOP-2IE63G4:~/ASSB/Practica_1/practical_proyectoGrupal$ ./calculoPI_par_prueba 10000000 12
El cálculo da 3.141908
aleingmar@DESKTOP-2IE63G4:~/ASSB/Practica_1/practical_proyectoGrupal$ ./calculoPI_par_prueba 100000000000 12
El cálculo da 3.141527
```

3. ACTIVIDADES

1. Ejecutar la aplicación para distintos valores del número de hilos midiendo los tiempos de ejecución, guardando en dos archivos de datos los resultados de todos los tiempos de ejecución y de la aceleración.

Para medir los tiempos de ejecución hemos ejecutado el programa del cálculo de pi en paralelo con 100000000 de lanzamientos para cada número de hilos desde 1 (tiempo de ejecución del programa en serie) hasta 2*N+2 hilos.

```
aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/Practica_1/practical_proyectoGrupal$ grep 'processor.*:' /proc/cpuinfo | wc -l
8
```

En el caso de nuestro ordenador, el número de procesadores lógicos es 8 y físicos 4 , por lo que hemos ejecutado el programa desde 1 hilo hasta 10.

```
aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/Practica_1/practical_proyectoGrupal$ /usr/bin/time -f"%e" ./calculoPI_par_prueba_2 100000000
0 1
El cálculo da 3.141485
"21.22"
aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/Practica_1/practical_proyectoGrupal$ /usr/bin/time -f"%e" ./calculoPI_par_prueba_2 100000000
0 2
El cálculo da 3.141602
"12.61"
aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/Practica_1/practical_proyectoGrupal$ /usr/bin/time -f"%e" ./calculoPI_par_prueba_2 100000000
0 3
El cálculo da 3.141572
"9.62"
aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/Practica_1/practical_proyectoGrupal$ /usr/bin/time -f"%e" ./calculoPI_par_prueba_2 100000000
0 4
El cálculo da 3.141507
"8.22"
aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/Practica_1/practical_proyectoGrupal$ /usr/bin/time -f"%e" ./calculoPI_par_prueba_2 100000000
0 5
El cálculo da 3.141531
"7.16"
aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/Practica_1/practical_proyectoGrupal$ /usr/bin/time -f"%e" ./calculoPI_par_prueba_2 100000000
0 6
El cálculo da 3.141540
"7.33"
```

```
aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/Practica_1/practica1_proyectoGrupal$ /usr/bin/time -f"%e" ./calculoPI_par_prueba_2 100000000
0 6
El cálculo da 3.141540
"7.33"
aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/Practica_1/practica1_proyectoGrupal$ /usr/bin/time -f"%e" ./calculoPI_par_prueba_2 100000000
0 7
El cálculo da 3.141574
"6.53"
aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/Practica_1/practica1_proyectoGrupal$ /usr/bin/time -f"%e" ./calculoPI_par_prueba_2 100000000
0 8
El cálculo da 3.141501
"5.83"
aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/Practica_1/practica1_proyectoGrupal$ /usr/bin/time -f"%e" ./calculoPI_par_prueba_2 100000000
0 9
El cálculo da 3.141509
"5.92"
aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/Practica_1/practica1_proyectoGrupal$ /usr/bin/time -f"%e" ./calculoPI_par_prueba_2 100000000
0 10
El cálculo da 3.141504
"5.93"
```

El resultado del tiempo de ejecución de la llamada a función para cada hilo se ha guardado en un fichero nano “mcpi_par-tiempo.dat” donde la primera columna guarda el numero de hilos y la segunda columna guarda el tiempo de ejecución.

```
  GNU nano 4.8                                    mcpi_par-tiempo.dat
1 21.22
2 12.61
3 9.62
4 8.22
5 7.61
6 7.33
7 6.53
8 5.83
9 5.92
10 5.99
```

De modo similar, se guarda la aceleración de la llamada a función para cada hilo en un fichero nano “mcpi_par-aceleracion.dat” donde la primera columna guarda el número de hilos y la segunda columna guarda la aceleración (calculada como la división entre el tiempo de ejecución del programa con un hilo y el tiempo de ejecución del programa con N hilos).

```
  GNU nano 4.8                                  mcpi_par-aceleracion.dat
1 1
2 1.68
3 2.20
4 2.58
5 2.79
6 2.90
7 3.25
8 3.64
9 3.58
10 3.54
```

2. Usando los archivos de datos, representar gráficamente el tiempo de ejecución y la aceleración obtenida en función del número de hilos.

Para representar gráficamente el tiempo de ejecución y la aceleración obtenida en función del número de hilos, hemos utilizado la herramienta de GNUPOT recomendada por el profesor, y ayudándonos en los archivos “aceleracion.gnp” y “tiempos.gnp” que se encuentran en la enseñanza virtual, hemos conseguido representar los datos de los ficheros.

Para obtener las gráficas correctamente, hemos tenido que modificar el fichero “aceleración.gnp” de la siguiente forma:

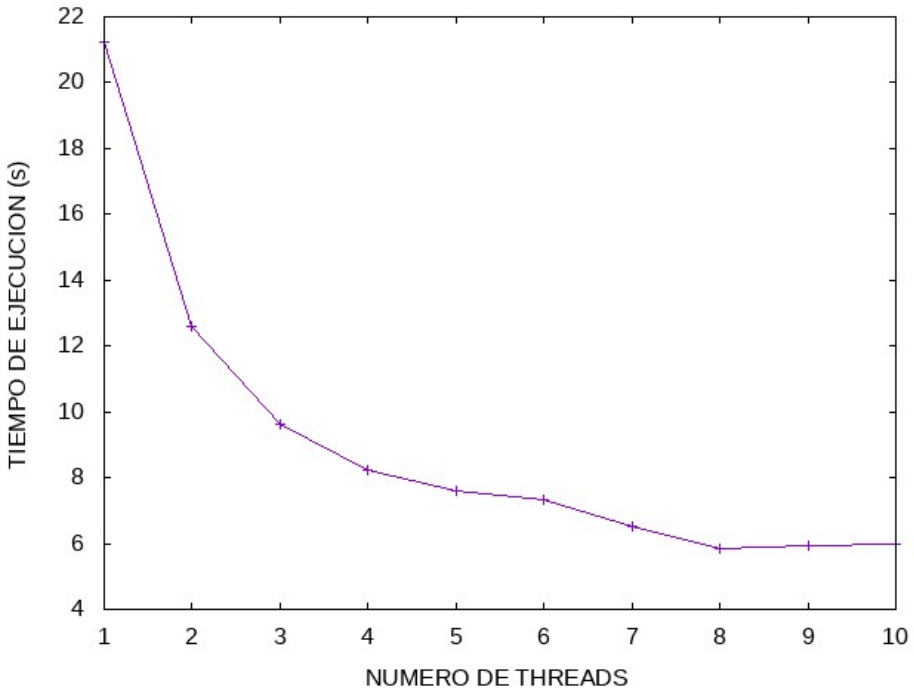
```
 GNU nano 4.8                                    aceleracion.gnp
# GRAFICA DE RESULTADOS USANDO GNUPLOT
reset
set terminal png
set output 'aceleracion.png'
set xlabel "NUM. DE PROCESADORES"
set ylabel "ACELERACION"

set xtics 5
set ytics 5

set xrange [0:]
set yrange [0:]

plot 'mcpi_par-aceleracion.dat' with linespoints title "ACELERACION EXPERIMENTAL", x title "ACELERACION LINEAL"
```

El fichero de “tiempos.gnp” no ha sido modificado.

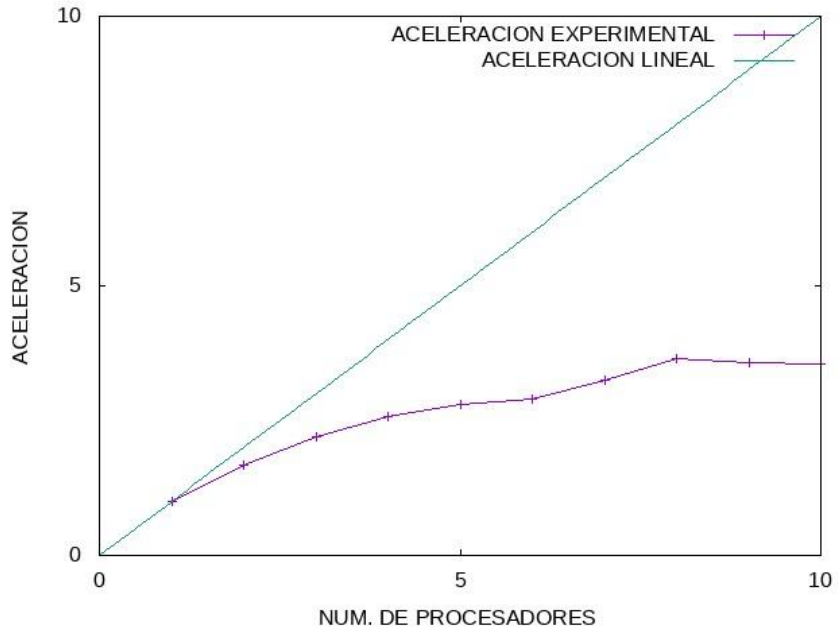
```
 GNU nano 6.2                                    tiempos.gnp *
# GRAFICA DE RESULTADOS USANDO GNUPLOT

reset
set terminal png
set output 'tiempos.png'
set xlabel "NUMERO DE THREADS"
set ylabel "TIEMPO DE EJECUCION (s)"
#set key graph 0.87,0.95 box

plot 'mcpi_par-tiempo.dat' notitle with linespoints
```

Finalmente, de la ejecución de los archivos anteriores se obtienen los siguientes resultados:

a) Gráfica del tiempo de ejecución según el número de hilos.



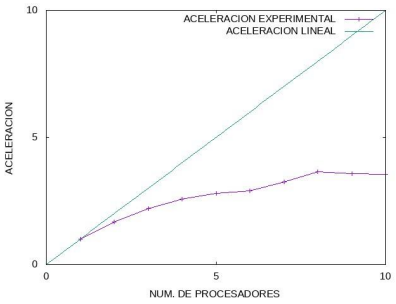
b) Gráfica de la aceleración según el número de hilos.



3. Interpretar los resultados analizando el rendimiento de la aplicación paralela, concretamente su aceleración y su escalabilidad, sobre el procesador con N núcleos.

Para interpretar los resultados obtenidos en el apartado 2 debemos tener en cuenta, en primer lugar, las características concretas del computador que está lanzando el programa.

En nuestro caso, el computador que ejecuta el programa tiene un procesador Intel Core i7 10th Generación, por lo que sabemos que tiene tecnología Hyper-Threading (HTT), es decir, que cada núcleo físico que posee el ordenador puede ejecutar dos instancias de hilos de ejecución. De esta forma, sabemos que nuestro ordenador tiene 4 cores físicos, pero 8 cores lógicos.

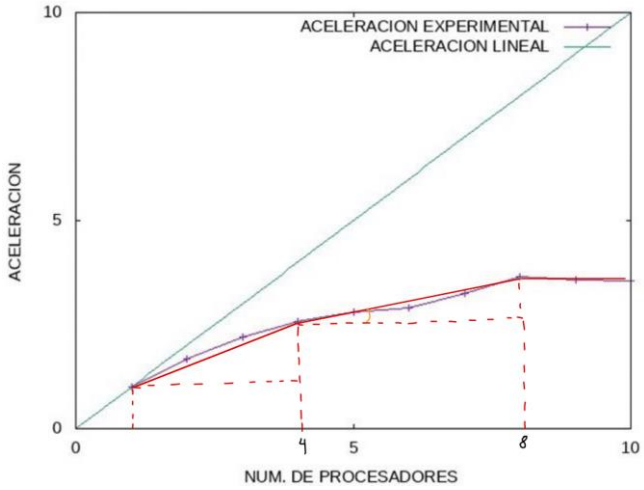



```
 GNU nano 4.8                                    mcpi_par-aceleracion.dat
1 1
2 1.68
3 2.20
4 2.58
5 2.79
6 2.90
7 3.25
8 3.64
9 3.58
10 3.54
```

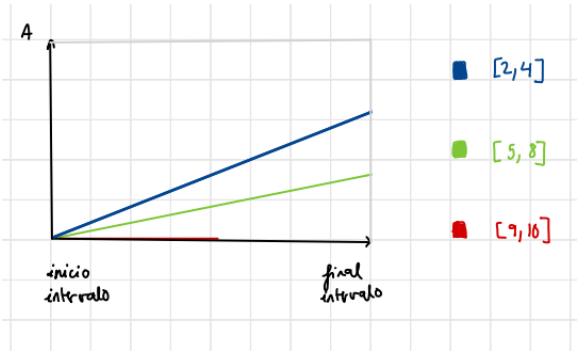
Para interpretar la gráfica de la aceleración vamos a dividir la gráfica en tres intervalos:

a) En el intervalo del número de hilos [2,4]
b) En el intervalo del número de hilos [5,8]
c) En el intervalo del número de hilos [9,10]

y vamos a representar geométricamente la pendiente de la recta desde el inicio de cada intervalo hasta el final de cada intervalo, para estudiar en mayor profundidad cómo se comporta el programa en cada intervalo.



Si extraemos de la gráfica la pendiente de la recta en cada intervalo y superponemos las pendientes en una misma gráfica, obtendríamos una segunda gráfica en la que se puede comparar mejor la aceleración por intervalo:



A partir de este punto podemos fácilmente interpretar que los mejores valores de aceleración se obtienen en el intervalo [2,4].

Sabiendo que el programa está paralelizado correctamente (evitando las condiciones de carrera mediante la técnica de la reducción). En términos de escalabilidad se puede decir que el programa es claramente escalable para el intervalo [2,4], es decir, cualitativamente el comportamiento del programa para desde 2 a 4 hilos es adecuado.

El siguiente intervalo [5,8] también presenta buenos valores de la recta de la aceleración en el intervalo, aunque de menor pendiente que en el intervalo primero. En términos de escalabilidad aún podemos confirmar que el programa escala correctamente para un número de hilos desde 5 hasta 8, pues la mejora de la aceleración es considerable aunque menor.

Por último, el intervalo [9,10] muestra un comportamiento contrapuesto al resto, la pendiente

de la recta es aproximadamente nula, es decir, la recta es prácticamente plana. Dentro del intervalo la aceleración apenas mejora al añadir mayor número de procesadores, y respecto al intervalo anterior los valores son incluso menores que los obtenidos previamente. Hay una degradación del rendimiento en términos de aceleración.

Esto se debe a que la CPU ya está utilizando todos los recursos disponibles y agregar hilos adicionales le resulta una sobrecarga de administración de hilos y competencia por recursos mucho mayor. Es decir, el computador se ve a crear uno o dos hilos extra, más allá de los hasta 8 hilos virtuales que ofrece el procesador, por lo que el ordenador "trabaja más", tiene que hacer un sobre-esfuerzo en crear mayor número de hilos del que verdaderamente está diseñado para proveer y coordinar cómo estos hilos acceden a los recursos. Por tanto, en términos de escalabilidad podemos decir que no es escalable para un número de hilos mayor o igual que 9.

De este análisis inferimos, que el cambio de pendiente de aceleración en los distintos intervalos analizados gráficamente se podría deber a que el primer intervalo se ejecuta con los 4 cores físicos, el segundo con los 4 cores lógicos reestantes y ya el resto con los 2 virtualizados.

Documentándonos hemos leído que es el gestor de tareas del SO el encargado de la repartición de recursos y de hilos en la ejecución del software, por lo que no está garantizado que esto sea así.