

Computación en la Nube

- **Nombre:** Pierre Simon Callist Yannick Tondreau
- **Repositorio Git:** https://github.com/PierreSimT/pr_cn/tree/master/p1
- **Máster Ingeniería Informática - Universidad de La Laguna**

Antes de realizar los ejercicios que están expuestos en la práctica se muestra el procesador sobre el cual se ejecutará los programas.

```
CPU:      Topology: Quad Core model: Intel Xeon E3-1270 V2 bits: 64 type: MT MCP
arch: Ivy Bridge family: 6 model-id: 3A (58) stepping: 9 microcode: 21
L2 cache: 8192 KiB
flags: avx lm nx pae sse sse2 sse3 sse4_1 sse4_2 ssse3 vmx bogomips: 55998
Speed: 1600 MHz min/max: 1600/3900 MHz Core speeds (MHz): 1: 1971 2: 1638 3: 1673
4: 1756 5: 1814 6: 1738 7: 2019 8: 1726
Vulnerabilities: Type: itlb_multihit status: KVM: Split huge pages
Type: l1tf mitigation: PTE Inversion; VMX: conditional cache flushes, SMT vulnerable
Type: mds mitigation: Clear CPU buffers; SMT vulnerable
Type: meltdown mitigation: PTI
Type: spec_store_bypass
mitigation: Speculative Store Bypass disabled via prctl and seccomp
Type: spectre_v1 mitigation: usercopy/swapgs barriers and __user pointer sanitization
Type: spectre_v2 mitigation: Full generic retpoline, IBPB: conditional, IBRS_FW,
STIBP: conditional, RSB filling
Type: tsx_async_abort status: Not affected
```

Ejercicio 1

Analiza el programa `hello.c` y realiza las siguientes ejecuciones comprobando en cada caso el resultado obtenido.

```
// hello.c
#include <stdio.h>
#include "mpi.h"

int main(int argc, char **argv) {
    int rank, size;
    int namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];

    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Get_processor_name(processor_name, &namelen);
    printf( "Hello world from process %d of %d in %s\n", rank, size,
processor_name );
    MPI_Finalize();
}
```

Análisis

A continuación se realizará una breve descripción de las funciones que aparecen en `hello.c`:

- **MPI_Init**: Inicializa la estructura de comunicación de *MPI* entre los procesos.
- **MPI_Comm_size**: Determina el tamaño del comunicador seleccionado, es decir, el número de procesos que están actualmente asociados a este.
- **MPI_Comm_rank**: Determina el rango (identificador) del proceso que lo llama dentro del comunicador seleccionado.
- **MPI_Finalize**: Finaliza la comunicación paralela entre los procesos.

También se puede apreciar las siguientes constantes:

- **MPI_COMM_WORLD**: Identificador del comunicador al que pertenecen todos los procesos de una ejecución *MPI*
- **MPI_MAX_PROCESSOR_NAME**: Valor que determina la longitud máxima del nombre que puede tener un procesador.

Con todo esto en cuenta, podemos deducir que lo único que realiza el programa al ejecutarse es mostrar un mensaje **Hello World** por cada proceso que ha instanciado en la ejecución.

Resultado

Al ejecutamos un solo proceso del programa obtenemos la siguiente salida:

```
> mpirun -np 1 hello.run
Hello world from process 0 of 1 in ptondreau
```

Si ejecutamos varios procesos del programa, por ejemplo 4, obtenemos la siguiente salida:

```
> mpirun -np 4 hello.run
Hello world from process 0 of 4 in ptondreau
Hello world from process 1 of 4 in ptondreau
Hello world from process 2 of 4 in ptondreau
Hello world from process 3 of 4 in ptondreau
```

Ejercicio 2

Analiza y compila el programa helloms.

```
// helloms.c
#include <stdio.h>
#include <string.h>
#include "mpi/mpi.h"

int main(int argc, char **argv) {
    int rank, size, tag, rc, i;
    MPI_Status status;
    char message[20];
    rc = MPI_Init(&argc, &argv);
    rc = MPI_Comm_size(MPI_COMM_WORLD, &size);
```

```
rc = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
tag = 100;
if(rank == 0) {
    strcpy(message, "Hello, world");
    for (i = 1; i < size; i++)
        rc = MPI_Send(message, 13, MPI_CHAR, i, tag, MPI_COMM_WORLD);
} else rc = MPI_Recv(message, 13, MPI_CHAR, 0, tag, MPI_COMM_WORLD,
&status);

printf( "node %d : %.13s\n", rank, message);
rc = MPI_Finalize();
}
```

Análisis

A parte de las mismas funciones y constantes que se han visto en el programa `hello.c`, se hace uso de los siguientes elementos:

- `MPI_Status`: Elemento que almacena la información sobre operaciones de recepción de mensajes.
- `MPI_Send`: Función que realiza el envío de un mensaje de un proceso fuente a otro destino.
- `MPI_Recv`: Rutina de recibimiento de un mensaje desde un proceso.

El programa `helloms.c` realiza lo siguiente:

1. Inicia las variables
2. Si el procesador es el procesador 0:
 - Almacena el mensaje `Hello, world` en la variable `message`.
 - Se envía el mensaje a cada procesador que esté disponible para el programa
3. En caso de no ser el procesador 0, espera para la recepción del mensaje
4. Muestra el mensaje en pantalla

Resultado

Ejecución del programa con 2 procesadores:

```
> mpirun -np 2 helloms.run
node 0 : Hello, world
node 1 : Hello, world
```

Ejecución del programa con 4 procesadores:

```
> mpirun -np 4 helloms.run
node 0 : Hello, world
node 2 : Hello, world
node 3 : Hello, world
node 1 : Hello, world
```

Ejercicio 3

Escribe un nuevo programa en el que los esclavos envían al maestro el mensaje y es el maestro el que muestra la salida.

Para realizar este ejercicio se usará el programa `helloms.c` como punto de partida y se modificará para cumplir lo que pide el ejercicio. El programa reescrito es el siguiente:

```
// helloms_ex.c
#include <stdio.h>
#include <string.h>
#include "mpi/mpi.h"

int main(int argc, char **argv) {
    int rank, size, tag, rc, i;
    MPI_Status status;
    char message[20];
    rc = MPI_Init(&argc, &argv);
    rc = MPI_Comm_size(MPI_COMM_WORLD, &size);
    rc = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    tag = 100;
    if(rank == 0) {
        for (i = 1; i < size; i++) {
            rc = MPI_Recv(message, 13, MPI_CHAR, i, tag, MPI_COMM_WORLD,
&status);
            printf( "node %d : %.13s\n", rank, message);
        }
    }
    else {
        strcpy(message, "Hello, world");
        rc = MPI_Send(message, 13, MPI_CHAR, 0, tag, MPI_COMM_WORLD);
    }

    rc = MPI_Finalize();
}
```

Resultado

Ejecución del programa con 2 procesadores:

```
> mpirun -np 2 helloms.run
node 0 : Hello, world
```

- Solo se muestra un mensaje dado que existe un maestro y un esclavo.

Ejecución del programa con 4 procesadores:

```
> mpirun -np 4 helloms.run
node 0 : Hello, world
node 0 : Hello, world
node 0 : Hello, world
```

- Se muestra tres mensajes dado que existen tres esclavos y un maestro.

Ejercicio 4

Escribe un programa que haga circular un token en un anillo

```
// token_ring.c
#include "mpi/mpi.h"
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv) {
    int rank, size, tag, rc, i;
    MPI_Status status;
    char message[20];

    rc = MPI_Init(&argc, &argv);
    rc = MPI_Comm_size(MPI_COMM_WORLD, &size);
    rc = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    tag = 100;

    if (size > 1) {
        if (rank == 0) {
            printf("Numero de procesadores a recorrer: %d\n", size);
            strcpy(message, "Hello, world");

            rc = MPI_Send(message, 13, MPI_CHAR, 1, tag, MPI_COMM_WORLD);

            printf("Maestro esperando que mensaje recorra el anillo\n");
            rc = MPI_Recv(message, 13, MPI_CHAR, size - 1, tag, MPI_COMM_WORLD,
                          &status);

            printf("node %d : %.13s\n", rank, message);
        } else {

            // Reciben mensaje por parte del procesador anterior
            rc = MPI_Recv(message, 13, MPI_CHAR, rank - 1, tag, MPI_COMM_WORLD,
                          &status);
            printf("Node %d ha recibido el mensaje : %.13s\n", rank, message);

            // Reenvia el mensaje al rango superior o al maestro
            if ((rank + 1) >= size) {
                rc = MPI_Send(message, 13, MPI_CHAR, 0, tag, MPI_COMM_WORLD);
            } else {
                rc = MPI_Send(message, 13, MPI_CHAR, rank + 1, tag,
```

```

MPI_COMM_WORLD);
    }
}
} else {
    printf("Numero de procesadores insuficiente\n");
}

rc = MPI_Finalize();
}

```

Resultado

Al ejecutar el programa con 4 procesadores la salida es la siguiente:

```

> mpirun -np 4 token.run
Numero de procesadores a recorrer: 4
Maestro esperando que mensaje recorra el anillo
Node 1 ha recibido el mensaje : Hello, world
Node 2 ha recibido el mensaje : Hello, world
Node 3 ha recibido el mensaje : Hello, world
node 0 : Hello, world

```

En cambio, si realizamos la ejecución con un solo procesador la salida es la siguiente:

```

> mpirun -np 1 token.run
Numero de procesadores insuficiente

```

Ejercicio 5

LOS PROGRAMAS DE ESTE EJERCICIO SE HAN EJECUTADO CON OTRO PROCESADOR

```

CPU:      Topology: Quad Core model: Intel Core i7-6700HQ bits: 64 type: MT MCP arch: Skylake-S rev: 3 L2 cache: 6144 KiB
flags: avx avx2 lm nx pae sse sse2 sse3 sse4_1 sse4_2 ssse3 vmx bogomips: 41599
Speed: 800 MHz min/max: 800/3500 MHz Core speeds (MHz): 1: 800 2: 800 3: 800 4: 800 5: 801 6: 800 7: 800 8: 800

```

El objetivo de este ejercicio es comprobar experimentalmente el costo de las comunicaciones entre pares de procesadores mediante ping-pong. Se trata además de comparar el coste de las comunicaciones con el coste de hacer una operación de tipo aritmético.

Programa `prod.c`

```

// prod.c
#include "mpi/mpi.h"
#include <stdio.h>
#include <math.h>

```

```

int main(int argc, char *argv[]) {
    int myid, numprocs;
    double startwtime, endwtime, proctime;
    int namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    double x, y, z;
    long int i, iterations;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Get_processor_name(processor_name,&namelen);

    fprintf(stdout,"Process %d of %d on %s\n", myid, numprocs,
processor_name);

    y = z = 1;
    iterations = 1000000000;

    startwtime = MPI_Wtime();
    for (i = 0; i < iterations; i++)
        x = y * z + x;
    endwtime = MPI_Wtime();

    proctime = (endwtime-startwtime) / (double) iterations;
    printf("wall clock time = %f, Prod time: %.16f, x = %f\n", (endwtime-
startwtime), proctime, x);

    MPI_Finalize();
    return 0;
}

```

La salida del programa `prod.c` nos mostrará el tiempo que se ha tardado para realizar un conjunto de operaciones, el tiempo que se ha tardado por operación y el número de operaciones que se han realizado.

En este caso, realiza 1000000000 operaciones aritméticas dadas por la función `x = y * z + x`

Resultado

La ejecución del programa se realiza con un único procesador y su salida es la siguiente:

```

> mpirun -np 1 prod.run
Process 0 of 1 on ptondreau-msi
wall clock time = 2.649659, Prod time: 0.0000000026496585, x =
1000000000.000000

```

Programa `ptop.c`

```

// ptop.c
#include <stdio.h>
#include <stdlib.h>
#include "mpi/mpi.h"

#define NUMBER_OF_TESTS 10

int main(int argc, char **argv) {
    double      *buf;
    int         rank;
    int         n;
    double      t1, t2, tmin;
    int         i, j, k, nloop;
    int my_name_length;
    char my_name[BUFSIZ];
    MPI_Status  status;

    MPI_Init( &argc, &argv );

    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Get_processor_name(my_name, &my_name_length);
    printf("\nProcesador: %s %d", my_name, rank);
    fflush(stdout);
    MPI_Barrier(MPI_COMM_WORLD);
    if (rank == 0)
        printf( "\nKind\t\t\t\t\ttime (sec)\t\tMB / sec\n" );

    for (n = 1; n < 34000000; n*=2) { // 1100000
        if (n == 0) nloop = 1000;
        else      nloop = 1000 / n;
        if (nloop < 1) nloop = 1;

        buf = (double *) malloc( n * sizeof(double) );
        if (!buf) {
            fprintf( stderr,
                    "Could not allocate send/recv buffer of size %d\n", n
);
            MPI_Abort( MPI_COMM_WORLD, 1 );
        }
        tmin = 1000;
        for (k = 0; k < NUMBER_OF_TESTS; k++) {
            if (rank == 0) {
                /* Make sure both processes are ready */
                MPI_Barrier(MPI_COMM_WORLD);
                t1 = MPI_Wtime();
                for (j = 0; j < nloop; j++) {
                    MPI_Send( buf, n, MPI_DOUBLE, 1, k, MPI_COMM_WORLD );
                    MPI_Recv( buf, n, MPI_DOUBLE, 1, k, MPI_COMM_WORLD,
&status );
                }
                t2 = (MPI_Wtime() - t1) / nloop;
                if (t2 < tmin) tmin = t2;
            }
        }
    }
}

```



```

        else if (rank == 1) {
            /* Make sure both processes are ready */
            MPI_Barrier(MPI_COMM_WORLD);
            for (j = 0; j < nloop; j++) {
                MPI_Recv( buf, n, MPI_DOUBLE, 0, k, MPI_COMM_WORLD,
&status );

                MPI_Send( buf, n, MPI_DOUBLE, 0, k, MPI_COMM_WORLD );
            }
        }
        /* Convert to half the round-trip time */
        tmin = tmin / 2.0;
        if (rank == 0) {
            double rate;
            if (tmin > 0) rate = n * sizeof(double) * 1.0e-6 /tmin;
            else
                rate = 0.0;
            printf( "Send/Recv\t%d\t%f\t%f\t\n", n, tmin, rate );
        }
        free( buf );
    }

    MPI_Finalize( );
    return 0;
}

```

El programa `ptop.c` realizará la comunicación de paso de mensajes entre dos procesadores. Realizará operaciones de envío y recepción de datos, dichas operaciones se realizarán con un número variable de tamaño de datos que aumenta a cada iteración, obtendrá el tiempo mínimo de cada iteración y finalmente mostrará por pantalla lo siguiente:

- El tamaño del elemento
- El tiempo que ha tardado en realizarse la operación
- La velocidad de la transmisión de datos

Resultado

La ejecución del programa se realiza con dos procesadores y la salida es la siguiente. A continuación se muestra el cálculo de la Regresión Lineal mediante LibreOffice Calc.

SE HA AUMENTADO EL NÚMERO MÁXIMO DE `n` DE 1048576 HASTA 33554432 PARA OBTENER MÁS DATOS

```
> mpirun -np 2 ptop.run
```

```

Procesador: ptondreau-msi 1
Procesador: ptondreau-msi 0
Kind      n   time (sec)  MB / sec
Send/Recv 1   0.000000    26.178953
Send/Recv 2   0.000000    52.770622
Send/Recv 4   0.000000   104.576530

```

Send/Recv	8	0.000000	204.248373
Send/Recv	16	0.000000	394.266832
Send/Recv	32	0.000000	777.962953
Send/Recv	64	0.000000	1304.126332
Send/Recv	128	0.000001	1397.815820
Send/Recv	256	0.000001	2873.713580
Send/Recv	512	0.000002	1855.494105
Send/Recv	1024	0.000003	3057.857047
Send/Recv	2048	0.000003	4799.766840
Send/Recv	4096	0.000005	7062.830319
Send/Recv	8192	0.000007	8918.888877
Send/Recv	16384	0.000012	10646.738413
Send/Recv	32768	0.000022	11697.374632
Send/Recv	65536	0.000041	12782.055927
Send/Recv	131072	0.000081	12937.476724
Send/Recv	262144	0.000163	12859.848807
Send/Recv	524288	0.000446	9404.817334
Send/Recv	1048576	0.001187	7064.721565
Send/Recv	2097152	0.002354	7125.889666
Send/Recv	4194304	0.004667	7189.640228
Send/Recv	8388608	0.009235	7266.420604
Send/Recv	16777216	0.018148	7395.748216
Send/Recv	33554432	0.036192	7416.990680

Generación de gráficas

```
# Importing the libraries
from sklearn.linear_model import LinearRegression
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

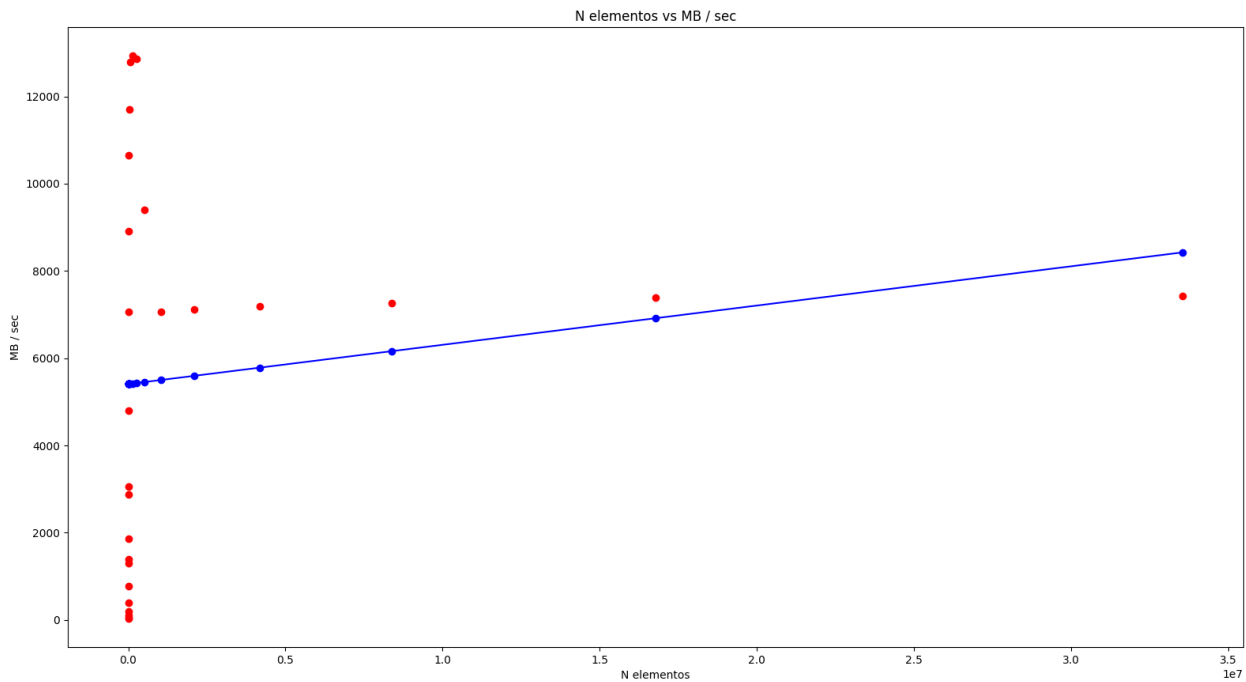
# Importing the dataset
dataset = pd.read_csv('data.csv')
X = dataset.iloc[:, 1:2].values
y = dataset.iloc[:, 3].values

# Fitting Simple Linear Regression to data
regressor = LinearRegression()
regressor.fit(X, y)

# Predicting the Test set results
y_prediction = regressor.predict(X)

# Visualising the Training set results
plt.scatter(X, y, color='red')
plt.scatter(X, y_prediction, color='blue')
plt.plot(X, y_prediction, color='blue')
plt.title('N elementos vs MB / sec')
plt.xlabel('N elementos')
```

```
plt.ylabel('MB / sec')
plt.show()
```



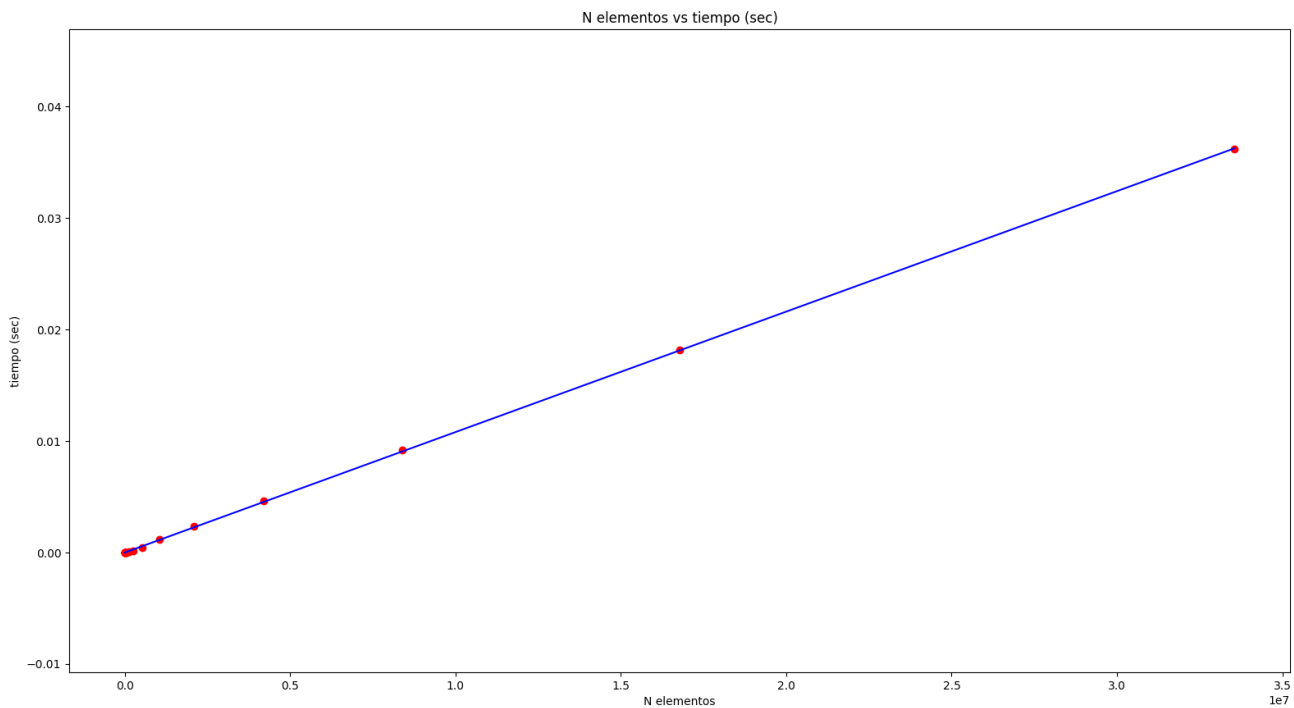
```
# Importing the libraries
from sklearn.linear_model import LinearRegression
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# Importing the dataset
dataset = pd.read_csv('data.csv')
X = dataset.iloc[:, 1:2].values
y = dataset.iloc[:, 2].values

# Fitting Simple Linear Regression to data
regressor = LinearRegression()
regressor.fit(X, y)

# Predicting the Test set results
y_prediction = regressor.predict(X)

# Visualising the Training set results
plt.scatter(X, y, color='red')
plt.plot(X, y_prediction, color='blue')
plt.title('N elementos vs tiempo (sec)')
plt.xlabel('N elementos')
plt.ylabel('tiempo (sec)')
plt.show()
```



Una vez calculada la regresión lineal, obtenemos los datos del regresor que se ha creado en Python:

```
Coeficientes:
[1.08064483e-09]
Mean squared error: 3.701446198421593e-09
Coeficiente de determinacion: 0.9999379475591803
```

Y la recta vendrá dada por la siguiente función:

Linear regression

$$y = 0.00000000108064483069x + 0.00000189054252995984$$

Conclusión

El coste de realizar una operación aritmética es de **0.0000000026496585** segundos, mientras que el coste de realizar la comunicación es de **0.000000** (no computable para un sólo dato) segundos para un dato de tipo **double**.

Por lo tanto, la conclusión que podemos sacar de esto es que realizar comunicaciones dentro del mismo procesador de core a core, se realizan muy rápidamente y su impacto en un programa puede ser mínimo. En cambio, cuando el número de datos aumenta, el coste de la comunicación comienza a ser mayor y puede empezar a notarse en la ejecución del programa.

De esta forma, se deberá tener en cuenta el número de datos que se enviarán en la comunicación dado que, dependiendo del problema, enviar un número mayor de datos permite reducir el costo de las comunicaciones ya que permite realizar operaciones sobre un rango mayor de datos.