PRÁCTICA 3. OPTIMIZACIÓN DE UN ALGORITMO GENÉTICO CON OPENMP Y MPI

ALBERTO GÁLVEZ GÁLVEZ | ANTONIO MARTINEZ FERNANDEZ | MARIO RODRÍGUEZ BÉJAR

Contenido

1. Algoritmo híbrido

1.1. Randr

Hacemos la variable *randomseed threadprivate* de manera que sea privada a cada hilo, lo que nos producirá diferentes números aleatorios en función del hilo que llame a la función. Sustituimos a su vez la llamada a *rand* por *rand_r* que es *threadprivate* para solucionar el problema mencionado anteriormente.

```
unsigned int randomSeed;
#pragma omp threadprivate(randomSeed)

static int aleatorio(int max)
{
        return (rand_r(&randomSeed) % (max + 1)); // rand_r en lugar de rand puesto que se usa esta funcion en paralelo
}
```

Ilustración 1. Declaración de la variable randomSeed y código de la función aleatorio con rand_r

Para que su uso sea correcto, creamos una sección paralela para que se inicialicen los diferentes hilos, cada uno de ellos inicializa su variable *randomseed* y entonces procede a inicializar la población y su fitness.

```
// Inicializamos la poblacion inicial y calculamos su fitness
#pragma omp parallel
{
    // Inicializar srandr
    randomSeed = 47 * time(NULL);
    #pragma omp for
    for(i = 0; i < tam_poblacion; i++) {
        init_imagen_aleatoria(poblacion[i].imagen, max, num_pixels);
        fitness(imagen_objetivo, &poblacion[i], num_pixels);
    }
}
```

Ilustración 2. Inicialización de la variable randomSeed

1.2. Código híbrido

Para conseguir una versión híbrida de *OMP* y *MPI* con el mejor rendimiento, hemos cogido la versión optimizada de *MPI* con el modelo de islas y le que le hemos introducido los siguientes cambios.

1.2.1. Introducción de directivas de paralelización de OMP

Tal como se observa en la ilustración 3, hemos añadido una sección paralela que engloba a la **inicialización de la población inicial**. Así, lo primero que se realizar en esta sección paralela es **inicializar** la variable privada de cada hilo ***randomSeed*** (para garantizar que la generación de valores aleatorios sea distinta en diferentes hilos). A continuación, lo que se realiza es una **paralelización** del **bucle** que **inicializa** cada una de las **imágenes** de la población inicial y después **calcula** su **fitness**.

```
// Inicializamos la poblacion inicial y calculamos su fitness
#pragma omp parallel
{
    // Inicializar srandr
    randomSeed = 47 * time(NULL);
    #pragma omp for
    for(i = 0; i < tam_poblacion; i++) {
        init_imagen_aleatoria(poblacion[i].imagen, max, num_pixels);
        fitness(imagen_objetivo, &poblacion[i], num_pixels);
    }
}
```

Ilustración 3. Inicialización de la población inicial con directivas OMP que realiza el proceso con id 0

Cabe destacar que **no** hemos **limitado** el **número de threads** que puede crear cada hilo, algo que a priori podría parecer que sobrecargaría el sistema. Sin embargo, para evitar este problema, hemos **añadido una barrera** (ilustración 4) justo después del código que ejecuta el padre, puesto que así el resto de los procesos quedan bloqueados esperando a que el proceso 0 termine de inicializar la población, así no habiendo un problema de sobrecarga (por ejemplo, si los demás procesos estuviesen realizando operaciones así consumiendo CPU). Además, todo esto lo hemos **comprobado experimentalmente**, para comprobar que nuestra hipótesis se confirmase.

```
// ********************************** INICIO TODOS ******************************************
MPI_Barrier(MPI_COMM_WORLD);
// Reservamos memoria para las estructuras de cada proceso
islaPoblacion = (Individuo *) malloc(chunkSize*sizeof(Individuo));
assert(islaPoblacion);
islaPoblacionNEM = (Individuo *) malloc(NEM*sizeof(Individuo));
assert(islaPoblacionNEM);
poblacionNPM = (Individuo *) malloc(NPM*sizeof(Individuo));
assert(poblacionNPM);

// El padre suministra subpoblaciones (de chunkSize) a los hijos
MPI_Barrier(MPI_COMM_WORLD);
MPI_Scatter(poblacion, chunkSize, typeIndividuo, islaPoblacion, chunkSize, typeIndividuo, 0, MPI_COMM_WORLD);

// ---------------- B. Evolucionar la Población (durante un número de generaciones) --------------------
for(int idNGM = 0; idNGM < num_generaciones/NGM; idNGM++) {
```

Ilustración 4. Código en el que se usa una barrera para que el padre disponga de tantos hilos como pueda

También, como se puede ver en la ilustración 5 y en la ilustración 6, se han usado directivas OMP para la **paralelización de los bucles de las funciones fitness y mutar**. Cabe mencionar que se han **limitado el número de hilos que puede crear cada proceso cuando llama a estas funciones a 2**, ya que de no limitarlo o poner un número mayor, el sistema se sobrecargaría, tal y como comprobamos experimentalmente (por ejemplo, si 6 procesos creasen 6 hilos en una máquina de 6 cores lógicos). Para la justificación de la elección de 2 hilos ver el apartado 1.2.2. Elección de valores de ejecución.

```
void fitness(const RGB *objetivo, Individuo *individuo, int num_pixels)
{
    double diff = 0.0;
    individuo->fitness = 0;

    #pragma omp parallel for reduction(+: diff) num_threads(2)
    for (int i = 0; i < num_pixels; i++)
    {
        diff +=abs(objetivo[i].r - individuo->imagen[i].r)
        + abs(objetivo[i].g - individuo->imagen[i].g)
        + abs(objetivo[i].b - individuo->imagen[i].b);
    }
    individuo->fitness = diff;
}
```

Ilustración 5. Función fitness con directivas OMP

```
void mutar(Individuo *actual, int max, int num_pixels)
{
    #pragma omp parallel for num_threads(2)
    for (int i = 0; i < num_pixels; i++)
    {
        if (aleatorio(1500)<= 1)
        {
            actual->imagen[i].r = aleatorio(max);
            actual->imagen[i].g = aleatorio(max);
            actual->imagen[i].b = aleatorio(max);
        }
    }
}
```

Ilustración 6. Función mutar con directivas OMP

1.2.2. Elección de valores de ejecución

En cuanto a los valores seleccionados para una ejecución óptima (vid ilustración 7) se ha realizado un estudio teórico y experimental con diversas combinaciones de los parámetros de ejecución del programa para encontrar los valores adecuados para un rendimiento óptimo.

Así, se ha probado a variar la distribución del número de procesos e hilos, siempre teniendo en cuenta que las máquinas en la que se ejecuta el programa son dos, una con 6 y la otra con 12 cores lógicos. Así, llegamos a la conclusión de que un número bajo de hilos por cada proceso es lo óptimo, suponiendo esto que el número de procesos ideal por cada máquina sea dividir el número de cores lógicos en ella entre el número de hilos por proceso. También, los parámetros propios de nuestra implementación del modelo de islas (NGM, NEM y NPM) han sido probados experimentalmente también, obteniendo una solución de compromiso entre la calidad de la solución (semejanza de la imagen de salida con la de entrada) y el rendimiento (tiempo que tarda en generarse la imagen de salida).

En cuanto a una justificación teórica que explique por qué lo anterior es lo óptimo, podríamos destacar lo siguiente:

- Si hubiese un número mayor de procesos y de hilos menor, el sobrecoste de la comunicación y sincronización sería mayor que los beneficios que se obtienen de esta, debido a que cada proceso y cada hilo no tendría una computación lo suficientemente grande.

- Si hubiese un número mayor de hilos y de procesos menor, el sobrecoste de la creación, comunicación, sincronización y destrucción de hilos sería mayor que los beneficios que se obtienen de esta, debido a que cada proceso no tendría una computación lo suficientemente grande.
- Si hubiese menos hilos por cada proceso o menos procesos, no se aprovecharían todos los cores y se obtendría un menor rendimiento.
- Si se aumentan el número de procesos y no se disminuye el de hilos, o viceversa, se produce un número mayor de hilos que de cores lógicos, lo que decrementa el rendimiento.
- Si aumenta el tamaño de las variables NGM, NEM o NPM el rendimiento disminuye. Si hacemos lo contrario, la solución es considerablemente peor.

```
T_POB = 256
N_GEN = 2000
NGM = 100
NEM = 4
NPM = 2

H1 = "marte"
NPH1 = 3
H2 = "venus"
NPH2 = 6

CC = mpicc
CFLAGS = -O3 -Wall -std=c99 -g -lm -fopenmp

EXEC = mpi
OUTFILE = ../output/out.txt
C_FILES = main.c imagen.c ga.c

mpi: $(C_FILES)
    $(CC) $(CFLAGS) $(C_FILES) -o $(EXEC) -DTIME -DDEBUG

test_mpi:
    mpirun \
    -np $(NPH1) -host $(H1) $(EXEC) \
    ../input/pitufo.ppm ../output/pitufo-out_$(N_GEN)_$(T_POB)_$(NGM)_$(NEM)_$(NPM).ppm \
    $(N_GEN) $(T_POB) $(NGM) $(NEM) $(NPM) \
    : -np $(NPH2) -host $(H2) $(EXEC) \
    ../input/pitufo.ppm ../output/pitufo-out_$(N_GEN)_$(T_POB)_$(NGM)_$(NEM)_$(NPM).ppm \
    $(N_GEN) $(T_POB) $(NGM) $(NEM) $(NPM) > $(OUTFILE)

clean:
    rm -f $(EXEC)
```

Ilustración 7. Makefile con las configuraciones de compilación y ejecución

2. Modelo de tiempo en memoria compartida

2.1. Paralelización bucle ini e instrumentalización

Para paralelizar el bucle de inicialización iniciamos la región paralela para que cada hilo tenga un valor del *randomseed* diferente y posteriormente en el bucle, con la directiva *pragma omp for* se inicializa el individuo y el fitness de este. Se especifica con *num_threads* el número de hilos que participarán en la ejecución del bucle.

```
    #pragma omp parallel
    {
            randomSeed = omp_get_thread_num() * time(NULL);
    }
    double ti = mseconds();
    #pragma omp parallel for num_threads(num_hilos_ini)
    for (i = 0; i < tam_poblacion; i++)
    {
            poblacion[i] = (Individuo *)malloc(sizeof(Individuo));
            poblacion[i]->imagen = imagen_aleatoria(max, num_pixels);
            fitness(imagen_objetivo, poblacion[i], num_pixels);
    }

    double tf = mseconds();
double time_taken = (tf - ti)/1000;
    printf("%f\n",time_taken);
    //printf("Con n_hilos_ini = %d - - - - tarda %d\n", num_hilos_ini, time_taken);
```

Ilustración 8. Inicialización con paralelización nivel 1

Para adaptar el código a nuestras necesidades, añadimos la medición del tiempo a través de las sentencias *mseconds,* la cual nos permite calcular el tiempo de ejecución del bucle FOR_INI en milisegundos.

2.2. Inclusión de n_hilos_ini n_hilos_fit

Se ha modificado la entrada del programa desde el *Makefile* y el *main.c* para poder aceptar los parámetros de número de hilos que participarán en la inicialización de los individuos.

```
  // Call Genetic Algorithm
  crear_imagen(imagen_objetivo, total, ancho, alto, max, \
                        num_generaciones, tam_poblacion, num_hilos_ini, num_hilos_fit, mejor_imagen, argv[2]);
```

Ilustración 9. Llamada a la función crear_imagen

```
void crear_imagen(const RGB *imagen_objetivo, int num_pixels, int ancho, int alto, int max, int num_generaciones, int tam_poblacion, int num_hilos_ini, int num_hilos_fit, RGB
{
```

Ilustración 10. Parámetros de entrada de la función crea_imagen con n_hilos_ini y n_hilos_fit

2.3. Pruebas bucle ini y análisis de resultados

Hemos probado el tiempo de ejecución del bucle inicializar con un número de hilos desde 2 a 18 aumentando el número de hilos en 2 cada vez e incluyendo el caso con un solo hilo. Los tiempos se pueden ver reflejados en la columna experimental.

```
#pragma omp parallel
{
        randomSeed = omp_get_thread_num() * time(NULL);
}
double ti = mseconds();
#pragma omp parallel for num_threads(num_hilos_ini)
for (i = 0; i < tam_poblacion; i++)
{
        poblacion[i] = (Individuo *)malloc(sizeof(Individuo));
        poblacion[i]->imagen = imagen_aleatoria(max, num_pixels);
        fitness(imagen_objetivo, poblacion[i], num_pixels);
}
```

Ilustración 11. Inicialización paralelizada con nivel 1

Se han plasmado en la tabla los tiempos obtenidos al ejecutar el algoritmo con la imagen proporcionada por la asignatura, con el número de iteraciones a 0 y los parámetros de la ecuación *NE* con valor 16. A partir de estos datos hemos aplicado el solucionador y hemos obtenido los siguientes datos:

Modelo de tiempo para Función con 1 Nivel de Paralelismo en Memoria Compartida

Modelo: $t_{un-nivel} = \frac{k_{s1} \cdot NE}{h} + k_h \cdot h$

NE = 16

Herramientas → Solucionador...

(Opciones → Algoritmo del Solucionador → DEPS Evolutionary Algorithm)

Constantes de Paralelismo:
k_{s1} = 0,032671883
k_h = 0,001284692

Cambiando las Celdas

Hilos	Tiempo		Ajuste
h	*experimental*	*teórico*	$(t\_exp - t\_teo)^2$
1	0,4940	0,5240	0,000902
2	0,2990	0,2639	0,001229
4	0,1380	0,1358	0,000005
6	0,1030	0,0948	0,000067
8	0,1010	0,0756	0,000644
10	0,0790	0,0651	0,000193
12	0,0880	0,0590	0,000842
14	0,0870	0,0553	0,001003
16	0,0650	0,0532	0,000139
18	0,0510	0,0522	0,000001
20	0,0600	0,0518	0,000067
22	0,0590	0,0520	0,000049
24	0,0670	0,0526	0,000207
26	0,0590	0,0535	0,000030
28	0,0580	0,0546	0,000011
30	0,0660	0,0560	0,000101
32	0,0650	0,0574	0,000057
34	0,0520	0,0591	0,000050
36	0,0560	0,0608	0,000023
38	0,0650	0,0626	0,000006
40	0,0620	0,0645	0,000006
42	0,0520	0,0664	0,000207
44	0,0600	0,0684	0,000071
46	0,0500	0,0705	0,000419
48	0,0620	0,0726	0,000111
	Suma		0,006439

Celda Objetivo → Mínimo

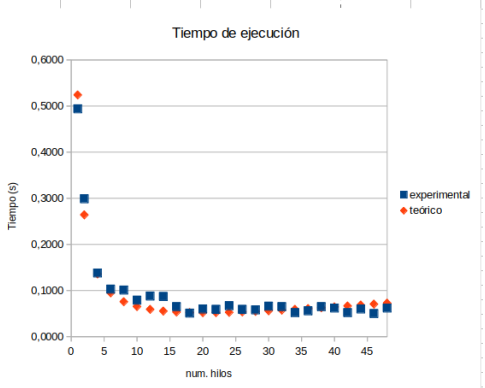


Ilustración 12. Diferencias entre tiempos experimentales y teóricos

Como podemos ver en la gráfica y en el valor suma de la tabla la función ajusta bien y la bondad de ajuste es correcta entre el tiempo teórico y experimental.

Entrando más en detalles podemos observar cómo gracias al solver obtenemos dos gráficas muy similares, lo que significa que nuestro modelo teórico se ajusta a nuestros tiempos experimentales.

Para un hilo tenemos la ejecución más lenta ya que, no se puede explotar la capacidad paralela del programa, a medida que aumentamos el número de hilos podemos ver una gran mejora hasta llegar a los 20 hilos, número en el cual la mejora tanto experimental como teórica decrece hasta el punto en el que no se puede apreciar mejora. Al añadir más de este número de hilos podemos ver saltos en los tiempos experimentales, pero en general los tiempos empeoran debido al exceso de sincronización entre tanta cantidad de hilos.

3. Ejecución del modelo de tiempo en memoria compartida

3.3. Calcular h_{opt} y tiempos experimentales

Para calcular el número de hilos óptimo debemos despejar la ecuación mostrada a continuación:

$$\frac{-k_{s1} \cdot NE}{h^2} + k_h = 0$$

Los pasos que hemos realizado para despejar la ecuación son los siguientes:

$$-ks1 \ * \ NE = -kh * h^2$$

$$\sqrt{\frac{-ks1 * NE}{-kh}} = h$$

Los valores usados han sido:

Ks1	0,032671883
kh	0,0012844692
NE	16

Tabla 1- constantes función FOR_INI

Resolviendo h en la ecuación obtenemos que el número óptimo de hilos es 19,3 hilos.

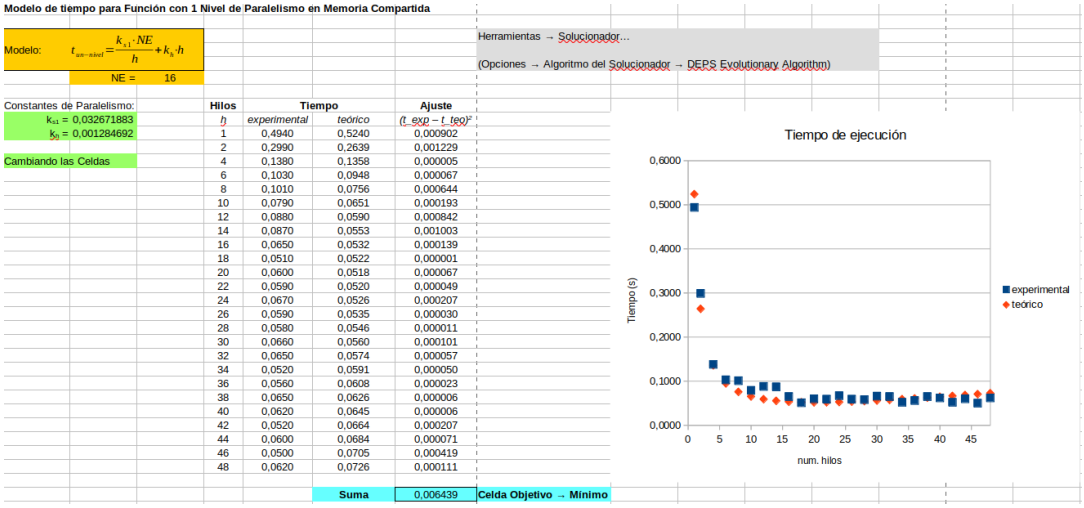


Ilustración 13. Tiempos teóricos y experimentales obtenidos previamente

En el apartado teórico hemos obtenido que el número de hilos con el cual se maximiza el rendimiento del programa es de 19 lo que contrasta perfectamente con los resultados experimentales donde podemos ver que a partir de 20 hilos no mejoran los tiempos o simplemente empeoran los resultados debido a la sincronización entre tanto número de hilos y la baja carga de trabajo repartida entre estos.

En la cuestión 3.3b, se nos solicita realizar la resolución de la fórmula teórica, y las constantes calculadas con anterioridad en el apartado 2.3, con un número diferente de hilos, por lo que hemos optado por calcular el tiempo de ejecución con 7, 15, 19, 21 y 33.

NÚMERO DE HILOS	TIEMPO DE EJECUCIÓN(s)
7	0,082
15	0,054
19	0,052
21	0,05255
33	0,060

Tabla 2- Valores teóricos obtenidos con los números de hilos seleccionados

3.3. Ejecución con NE=16 y bondad de ajuste

En este apartado se nos pide que ejecutemos nuestro algoritmo con los hilos seleccionados en el apartado anterior y con un valor de NE=16 en el clúster, así como, calcular la bondad de ajuste con la funcionalidad proporcionada por el Excel.

En cuanto a los tiempos de ejecución obtenidos experimentalmente en el Cluster obtenemos los visibles en la tabla 3.

NÚMERO DE HILOS	TIEMPO DE EJECUCIÓN(s)
7	0,1120
15	0,0890
19	0,0560
21	0,0640
33	0,0540

Tabla 3. Valores experimentales obtenidos para los números de hilos seleccionados

Al poner los datos experimentales calculados en la hoja de cálculo podemos observar en la ilustración 14 que hemos obtenido una bondad de ajuste de 0,000041, lo que nos indica que hemos obtenido unos tiempos en el clúster consecuentes con lo calculado teóricamente anteriormente y que las constantes obtenidas son extrapolables a otros números de hilos.

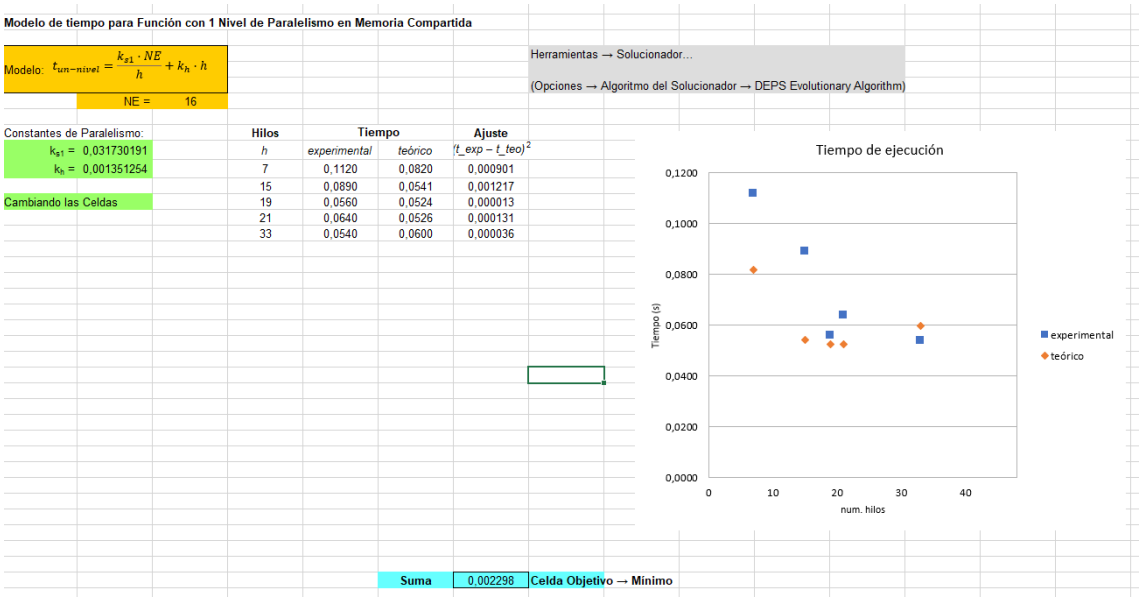


Ilustración 14. Tiempos experimentales

3.3. Tiempo ejecución con diferentes NE

Volvemos a resolver la ecuación:

$$\frac{-k_{s1} \cdot NE}{h^2} + k_h = 0$$

Pero esta vez cambiando los valores de NE y obtenemos los siguientes números de hilos óptimos:

NE	N hilos óptimo
12	16.78643
20	21.67119
30	26.54168
40	30.64769

Tabla 4- Valores de hilos óptimos obtenidos en función del valor de NE

3.3. Bondad de ajuste con NE=12 y 20

Al ejecutar con NE=12 obtenemos una bondad de ajuste de 0,005197, los tiempos experimentales son los siguientes de la columna tiempo experimental. Podemos observar que el modelo ajusta correctamente, mejora significativamente hasta un número de hilos de 22 y a partir de ahí no mejoran los valores sustancialmente debido a la sincronización entre un gran número de hilos.

Modelo de tiempo para Función con 1 Nivel de Paralelismo en Memoria Compartida

Modelo: $t_{un-nivel} = \frac{k_{s1} \cdot NE}{h} + k_h \cdot h$

NE = 12

Herramientas → Solucionador…

(Opciones → Algoritmo del Solucionador → DEPS Evolutionary Algorithm)

Constantes de Paralelismo:

k_{s1} = 0,032671883

k_0 = 0,001284692

Cambiando las Celdas

Hilos	Tiempo		Ajuste
h	*experimental*	*teórico*	*(t_exp – t_teo)²*
1	0,3720	0,3933	0,000456
2	0,1950	0,1986	0,000013
4	0,1070	0,1032	0,000015
6	0,0770	0,0731	0,000016
8	0,0840	0,0593	0,000611
10	0,0930	0,0521	0,001677
12	0,0660	0,0481	0,000321
14	0,0660	0,0460	0,000400
16	0,0570	0,0451	0,000143
18	0,0620	0,0449	0,000292
20	0,0590	0,0453	0,000188
22	0,0450	0,0461	0,000001
24	0,0530	0,0472	0,000034
26	0,0550	0,0485	0,000042
28	0,0600	0,0500	0,000101
30	0,0510	0,0516	0,000000
32	0,0540	0,0534	0,000000
34	0,0530	0,0552	0,000005
36	0,0540	0,0571	0,000010
38	0,0550	0,0591	0,000017
40	0,0640	0,0612	0,000008
42	0,0670	0,0633	0,000014
44	0,0500	0,0654	0,000238
46	0,0590	0,0676	0,000074
48	0,0470	0,0698	0,000521
	Suma		0,005197

Celda Objetivo → Mínimo

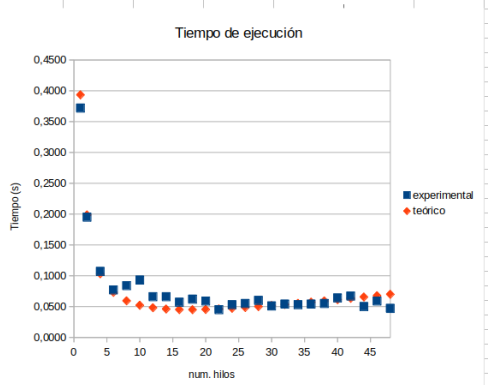


Ilustración 15. Tiempos experimentales con NE=12

Al ejecutar con NE=20 obtenemos una bondad de ajuste de 0,007074.

Modelo de tiempo para Función con 1 Nivel de Paralelismo en Memoria Compartida

Modelo: $t_{un-nivel} = \frac{k_{s1} \cdot NE}{h} + k_h \cdot h$

NE = 20

Herramientas → Solucionador…

(Opciones → Algoritmo del Solucionador → DEPS Evolutionary Algorithm)

Constantes de Paralelismo:

k_{s1} = 0,032671883

k_0 = 0,001284692

Cambiando las Celdas

Hilos	Tiempo		Ajuste
h	*experimental*	*teórico*	*(t_exp – t_teo)²*
1	0,6170	0,6547	0,001423
2	0,3180	0,3293	0,000127
4	0,1650	0,1685	0,000012
6	0,1370	0,1166	0,000416
8	0,1080	0,0920	0,000257
10	0,1090	0,0782	0,000949
12	0,0770	0,0699	0,000051
14	0,1000	0,0647	0,001249
16	0,0910	0,0614	0,000876
18	0,0880	0,0594	0,000816
20	0,0580	0,0584	0,000000
22	0,0640	0,0580	0,000036
24	0,0620	0,0581	0,000016
26	0,0630	0,0585	0,000020
28	0,0590	0,0593	0,000000
30	0,0570	0,0603	0,000011
32	0,0600	0,0615	0,000002
34	0,0670	0,0629	0,000017
36	0,0800	0,0644	0,000243
38	0,0730	0,0660	0,000049
40	0,0660	0,0677	0,000003
42	0,0590	0,0695	0,000111
44	0,0630	0,0714	0,000070
46	0,0660	0,0733	0,000053
48	0,0590	0,0753	0,000265
	Suma		0,007074

Celda Objetivo → Mínimo

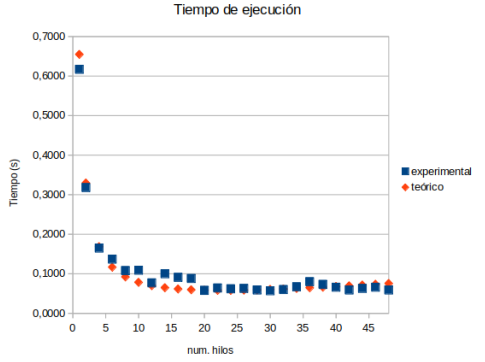


Ilustración 16. Tiempos experimentales con NE=20

Obtenemos también un buen ajuste y la diferencia más notable es que la caída de rendimiento con NE=20 se produce a partir de 20 hilos como se refleja tanto en los tiempos teóricos como experimentales.

4. Modelo de tiempo en memoria compartida

4.1. Instalación rutina con 2 niveles de paralelismo

Para implementar la paralelización en la función fitness hay que especificar la directriz omp_set_nested(1) para indicarle a omp que se va a realizar un paralelismo anidado, el cual se llevara a cabo en la función **fitness**, los demás cambios realizados ya han sido explicados en el apartado X.

```
    #pragma omp parallel
    {
            randomSeed = omp_get_thread_num() * time(NULL);
    }
    double ti = mseconds();
    omp_set_nested(1);
    #pragma omp parallel for num_threads(num_hilos_ini)
    for (i = 0; i < tam_poblacion; i++)
    {
            poblacion[i] = (Individuo *)malloc(sizeof(Individuo));
            poblacion[i]->imagen = imagen_aleatoria(max, num_pixels);
            fitness(imagen_objetivo, poblacion[i], num_pixels,num_hilos_fit);
    }

    double tf = mseconds();
double time_taken = (tf - ti)/1000;
    printf("%f\n",time_taken);
```

Ilustración 17. Código paralelizado con nivel 2

Dentro de la función fitness simplemente elegimos la directriz que nos ha reportado el mejor rendimiento en programas anteriores, incluyéndole el parámetro **num_threads** con el número de hilos especificado para esta función, la cual se ve representada por el parámetro **num_hilos_fit**.

```
void fitness(const RGB *objetivo, Individuo *individuo, int num_pixels, int num_hilos_fit)
{
        // Determina la calidad del individuo (similitud con el objetivo)
        // calculando la suma de la distancia existente entre los pixeles

        double diff = 0.0;

        // TODO ejecutar los 3 por separado

        individuo->fitness = 0;


        #pragma omp parallel for reduction(+: diff) num_threads(num_hilos_fit)
        for (int i = 0; i < num_pixels; i++)
        {
                diff +=abs(objetivo[i].r - individuo->imagen[i].r) + abs(objetivo[i].g - individuo->imagen[i].g) + abs(objetivo[i].b - individuo->imagen[i].b);
        }

        individuo->fitness=diff;
}
```

Ilustración 18. Función fitness paralelizada internamente con num_threads

Luego, tras realizar los cambios necesarios en el código, simplemente nos queda realizar las distintas ejecuciones necesarias para comprobar que nuestro modelo se ajusta a los tiempos experimentales, para ello hemos decidido comprobar todas las combinaciones para los números de hilos 1, 2, 4, 8, 16, 24, 48 para **h1** y **h2**.

Una vez obtenidos los tiempos de ejecuciones inicialmente hemos decidido ajustar todos estos casos con las mismas constantes **kh1, kh2, ks2** para todas las ejecuciones, obteniendo como resultado lo que podemos ver en la ilustración 19.

Modelo de tiempo para Función con 2 Niveles de Paralelismo en Memoria Compartida

Modelo: $t_{dos-niveles} = \frac{k_{s2} \cdot NE \cdot NP}{h_1 \cdot h_2} + k_{h,1} \cdot h_1 + k_{h,2} \cdot h_2$

NE = 16
NP = 1048576

Constantes de Paralelismo:
k_{s2} =4,25793E-08
$k_{h,1}$ =1,23081E-16
$k_{h,2}$ = 0,00837787

Cambiando las Celdas

Hilos		Tiempo		Ajuste
h1	*h2*	*experimental*	*teórico*	*(t_exp – t_teo)²*
1	1	0,498	0,72	0,0505079425
1	2	0,513	0,37	0,019338609
1	4	0,483	0,21	0,073385748
1	8	0,509	0,16	0,1243844327
1	16	0,512	0,18	0,1110931506
1	24	0,531	0,23	0,0900995876
1	48	0,541	0,42	0,015370912
2	1	0,256	0,37	0,0120031303
2	2	0,297	0,20	0,0103334928
2	4	0,24	0,12	0,0137342627
2	8	0,256	0,11	0,0208309722
2	16	0,358	0,16	0,0406547372
2	24	0,392	0,22	0,0309930645
2	48	0,293	0,41	0,0135907212
4	1	0,14	0,19	0,0022060246
4	2	0,139	0,11	0,0010856379
4	4	0,149	0,08	0,0050184312
4	8	0,174	0,09	0,0071661635
4	16	0,183	0,15	0,0014282431
4	24	0,18	0,21	0,0008128349
4	48	0,227	0,41	0,0319904037
8	1	0,082	0,10	0,0002456462
8	2	0,092	0,06	0,0009361541
8	4	0,098	0,06	0,0017778614
8	8	0,11	0,08	0,0010122006
8	16	0,117	0,14	0,0005119786
8	24	0,191	0,20	0,0001901537
8	48	0,345	0,40	0,003480799
16	1	0,058	0,05	2,474575E-05
16	2	0,056	0,04	0,0002863014
16	4	0,062	0,04	0,000300211
16	8	0,057	0,07	0,0002434832
16	16	0,097	0,14	0,0015869443
16	24	0,194	0,20	7,9732522E-05
16	48	0,378	0,40	0,0006284115
24	1	0,065	0,04	0,0007213011
24	2	0,064	0,03	0,0010472804
24	4	0,06	0,04	0,0003627971
24	8	0,095	0,07	0,0005883714
24	16	0,093	0,14	0,0018409515
24	24	0,123	0,20	0,0062899494
24	48	0,341	0,40	0,0038140611
48	1	0,088	0,02	0,0041912141
48	2	0,048	0,02	0,0005665819
48	4	0,065	0,04	0,0007710544
48	8	0,06	0,07	7,8913265E-05
48	16	0,167	0,13	0,0010255269
48	24	0,153	0,20	0,002370628
48	48	0,246	0,40	0,0244759867
			suma	0,735477743

Ilustración 19. Comparación tiempos experimentales y teóricos

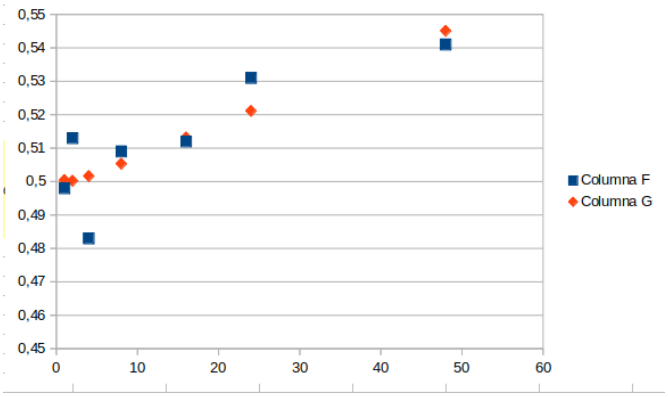
Debido al mal ajuste de nuestro modelo con los datos experimentales obtenidos, hemos decidido optar por aislar varios conjuntos de ejecuciones según el valor de **h1** y obtener constantes **kh1, kh2 y ks2** específicas para cada conjunto de forma que se obtenga un ajuste más cercano.

Hemos probado para los valores de h1 {1,2,4,8,16,24,48} y estos han sido los resultados:

H1 = 1

Constantes de Paralelismo:		Hilos		Tiempo		Ajuste
k_{s2} = 1,44961E-10		*h1*	*h2*	*experimental*	*teórico*	$(t\_exp - t\_teo)^2$
k_{h1} = 0,497010998		1	1	0,498	0,50	5,97123379E-06
k_{h2} = 0,001000572		1	2	0,513	0,50	0,000163119825
		1	4	0,483	0,50	0,000346752704
Cambiando las Celdas		1	8	0,509	0,51	1,35454868E-05
		1	16	0,512	0,51	1,37394812E-06
		1	24	0,531	0,52	9,74945913E-05
		1	48	0,541	0,55	1,67209778E-05

Ilustración 20 Datos obtenidos para el valor de num_hilos_ini 1



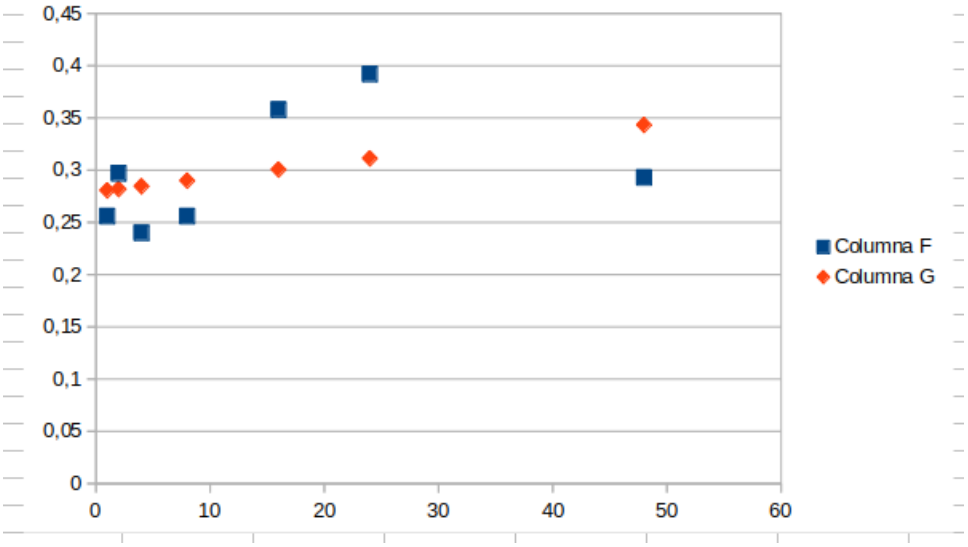
Suma-1	0,000644978767

Ilustración 21. Datos obtenidos para el valor de num_hilos_ini 1

H1 = 2

Constantes de Paralelismo:		2	1	0,256	0,28	0,000603040219
k_{s2} = 1,10044E-20		2	2	0,297	0,28	0,000228273705
k_{h1} = 0,139611242		2	4	0,24	0,28	0,001985598703
k_{h2} = 0,001334393		2	8	0,256	0,29	0,001149049363
		2	16	0,358	0,30	0,003297885896
		2	24	0,392	0,31	0,006520897754
		2	48	0,293	0,34	0,002527411171

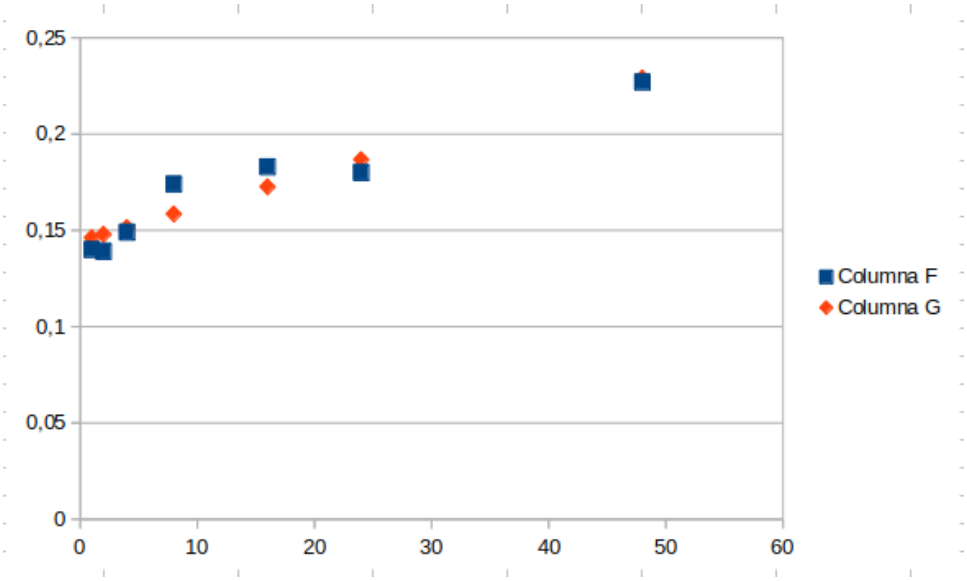
Ilustración 22 Datos obtenidos para el valor de num_hilos_ini 2



Suma-2	0,01631215681

Ilustración 23. Datos obtenidos para el valor de num_hilos_ini 2

H1 = 4



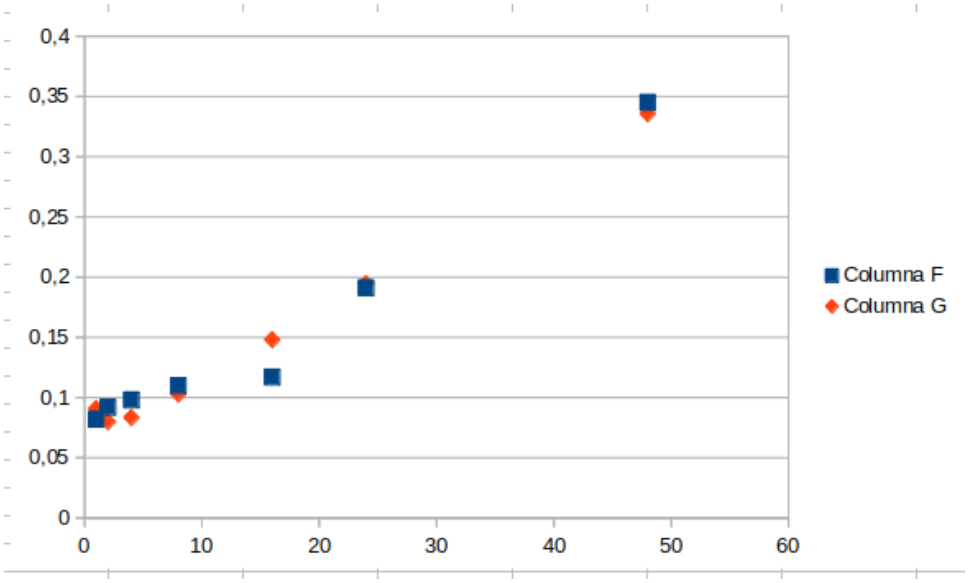
Suma-4	0,000520905009

Ilustración 24. Datos obtenidos para el valor de num_hilos_ini 4

H1 = 8

Constantes de Paralelismo:		8	1	0,082	0,09	7,58026401E-05
k_{s2} = 1,59276E-08		8	2	0,092	0,08	0,000146116143
k_{h1} = 0,006424621		8	4	0,098	0,08	0,000213878439
k_{h2} = 0,005906953		8	8	0,11	0,10	5,14389032E-05
		8	16	0,117	0,15	0,000960743887
		8	24	0,191	0,19	1,26423098E-05
		8	48	0,345	0,34	8,78610648E-05

Ilustración 25Datos obtenidos para el valor de num_hilos_ini 8



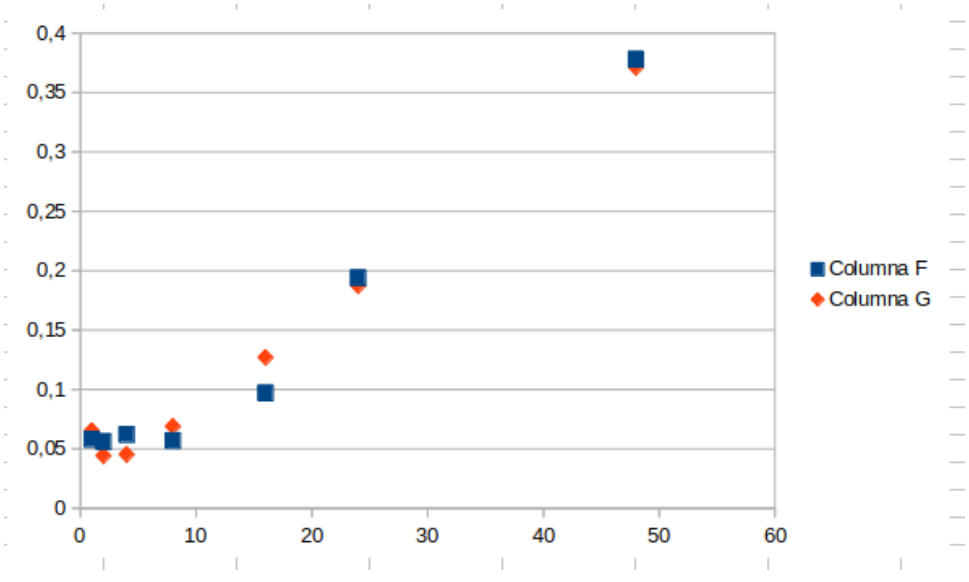
Suma-8	0,001548483387

Ilustración 26. Datos obtenidos para el valor de num_hilos_ini 8

H1 = 16

Constantes de Paralelismo:	16	1	0,058	0,07	5,17400262E-05
k_{s2} = 5,48232E-08	16	2	0,056	0,04	0,000140263192
k_{h1} = 2,36029E-14	16	4	0,062	0,05	0,000282281821
k_{h2} = 0,007706798	16	8	0,057	0,07	0,0001401895
	16	16	0,097	0,13	0,000894109051
	16	24	0,194	0,19	4,41107358E-05
	16	48	0,378	0,37	4,728039E-05

Ilustración 27 Datos obtenidos para el valor de num_hilos_ini 16



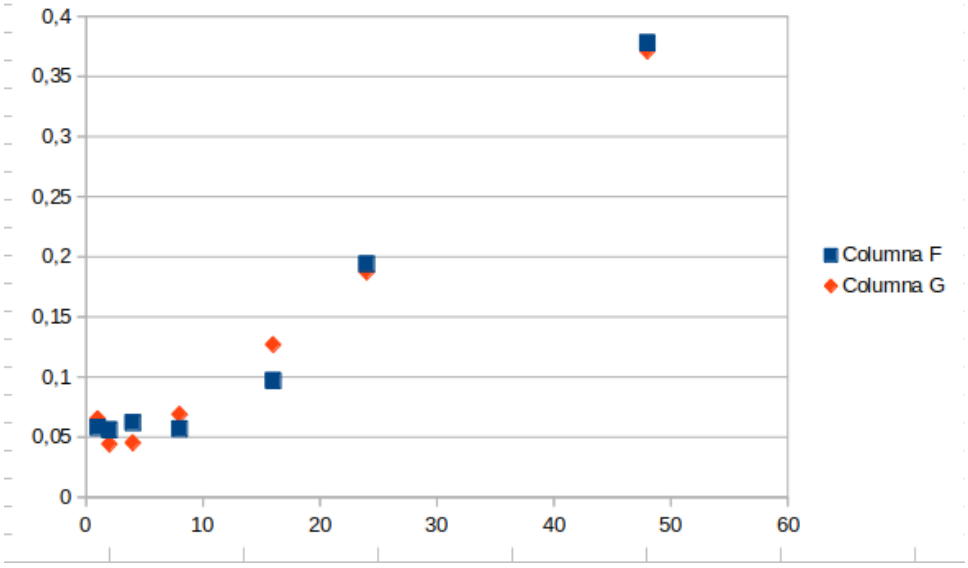
Suma-16	0,001599974715

Ilustración 28. Datos obtenidos para el valor de num_hilos_ini 16

H1 = 24

Constantes de Paralelismo:		24	1	0,065	0,07	9,90274345E-05
k_{s2} = 8,63213E-08		24	2	0,064	0,05	0,000163383236
k_{h1} = 0,000340425		24	4	0,06	0,05	0,000120817877
k_{h2} = 0,006438084		24	8	0,095	0,07	0,000771853628
		24	16	0,093	0,11	0,000481845582
		24	24	0,123	0,17	0,001780713967
		24	48	0,341	0,32	0,000508260212

Ilustración 24 Datos obtenidos para el valor de num_hilos_ini



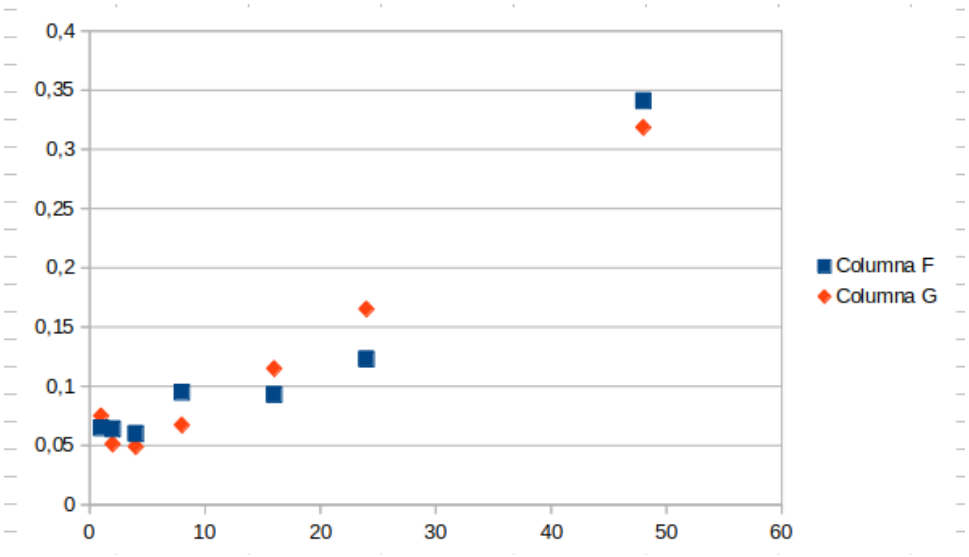
Suma-24	0,003925901937

Ilustración 29. Datos obtenidos para el valor de num_hilos_ini 24

H1 = 48

Constantes de Paralelismo:		48	1	0,088	0,08	0,00013066416
k_{s2} = 7,48662E-08		48	2	0,048	0,07	0,000395200339
k_{h1} = 0,000958483		48	4	0,065	0,07	2,62795854E-05
k_{h2} = 0,004394314		48	8	0,06	0,08	0,000596954956
		48	16	0,167	0,12	0,002405736146
		48	24	0,153	0,15	1,92678245E-07
		48	48	0.246	0.26	0.000131777225

Ilustración 30 Datos obtenidos para el valor de num_hilos_ini 48



Suma-48	0,003686805089

Tras observar estos valores y gráficas podemos ver que el ajuste al modelo teórico es bastante bueno, encontrando un buen rendimiento a la hora de usar en conjunto un bucle exterior con muchos hilos y un bucle interno con un numero de hilos reducido, ya que la carga que encontramos en ese bucle es baja y el uso de hilos en esta por cada uno de los hilos del bucle externo supone una gran carga de trabajo.

4.2. Hilos óptimos teóricos

Primero partimos de este sistema de ecuaciones en el cual debemos de despejar h1 y h2 para obtener la combinación de hilos optima.

$$\frac{-Ks2 * NE * NP}{h1^2 * h2} + kh1 = 0$$

$$\frac{-Ks2 * NE * NP}{h2^2 * h1} + kh2 = 0$$

El primer paso sería multiplicar todo por $h1^2 * h2$ y $h2^2 * h1$ en sus respectivas ecuaciones para quitarnos el denominador.

1. $-Ks2 * NE * NP + kh1 * h1^2 * h2 = 0$
2. $-Ks2 * NE * NP + kh2 * h2^2 * h1 = 0$

Luego, vamos a aplicar el método de sustitución por lo que despejamos de la ecuación **1** el **h1**, de esta forma podremos utilizar este valor para sustituirlo en la ecuación **2**, además, este valor nos valdrá para obtener el valor de **h1** posteriormente tras calcular **h2**.

$$h1 = \sqrt{\frac{Ks2 * NE * NP}{kh1 * h2}}$$

Después, eliminamos el negativo de ambos lados y sustituimos **h1** por el valor anterior.

$$Ks2 * NE * NP = kh2 * h2^2 * \sqrt{\frac{Ks2 * NE * NP}{kh1 * h2}}$$

A continuación, elevamos al cuadrado ambas partes para quitarnos la raíz cuadrada.

$$(Ks2 * NE * NP)^2 = kh2^2 * h2^4 * \frac{Ks2 * NE * NP}{kh1 * h2}$$

Simplificamos para quitarnos elementos del denominador.

$$(Ks2 * NE * NP)^2 - \frac{kh2^2 * h2^3 * Ks2 * NE * NP}{kh1} = 0$$

Despejamos $h2^3$.

$$\frac{(Ks2 * NE * NP)^2 * kh1}{kh2^2 * NE * NP * Ks2} = h2^3$$

Simplificamos los valores del numerador y denominador.

$$\frac{Ks2 * NE * NP * kh1}{kh2^2} = h2^3$$

Y finalmente obtenemos la expresión que nos da el valor de h2.

$$\sqrt[3]{\frac{Ks2 * NE * NP * kh1}{kh2^2}} = h2$$

Tras despejar y obtener las expresiones de las que podemos obtener el valor de h1 y h2 podemos realizar los cálculos, obteniendo los siguientes valores:

Constantes	Hilos 1 óptimo	Hilos 2 óptimo
h1 = 1	0.02 -> 1	10.65 -> 10
h1 = 2	0.000023 -> 1	0.0024 -> 1
h1 = 4	0.000119 -> 1	0.0024 -> 1
h1 = 8	3.388 -> 3	3.625 -> 3
h1 = 16	2.33 -> 2	0.000715 -> 1
h1 = 24	43.170 -> 43	2.28 -> 2
h1 = 48	183.235 -> 183	0.04 -> 1

Tabla 5. Hilos óptimos 2 niveles de paralelización.

En la tabla 5 podemos ver como los valores de los hilos óptimos no se ajustan en la mayor parte de los casos a los resultados obtenidos de forma experimental ni a los teóricos exceptuando los valores obtenidos de **h1=1**, donde vemos que se ajusta al valor teórico esperado, o el **h1=24**, en el cual se ajusta a un valor muy cercano al óptimo obtenido en la tabla de tiempos para la función con 2 niveles de paralelismo que es el **48 2**.

Estos malos resultados, pueden deberse a que el valor teórico y experimental no ajustan muy bien por lo que las constantes obtenidas no se asemejan a un valor realista dando lugar a resultados erróneos.

4.3. Ejecuciones con diferentes H1 y H2

Hemos realizado los cálculos en la hoja de cálculo con los parámetros usados para el apartado 4 para obtener los diferentes tiempos de ejecución experimentales. Los valores que vamos a variar son NE y NP.

Como podemos observar los tiempos no varían prácticamente nada, aunque modifiquemos ambos valores drásticamente. Esto es debido a que al ser las otras constantes un número muy pequeño al ser multiplicadas por NE y NP en la ecuación realmente no se nota el impacto de estos números.

NE=16, NP=1048576

Modelo de tiempo para Función con 2 Niveles de Paralelismo en Memoria Compartida

Modelo: $t_{dos-niveles} = \frac{k_{s2} \cdot NE \cdot NP}{h_1 \cdot h_2} + k_{h,1} \cdot h_1 + k_{h,2} \cdot h_2$

NE = 16
NP = 1048576

Constantes de Paralelismo:	Hilos		Tiempo
k_{s2} = 4,25793E-08	*h1*	*h2*	*teórico*
k_{h1} = 1,23081E-16	12	12	0,11
k_{h2} = 0,008377875	12	32	0,27
	12	36	0,30
Cambiando las Celdas	32	12	0,10
	32	32	0,27
	32	36	0,30
	36	12	0,10
	36	32	0,27
	36	36	0,30

Ilustración 31. NE=16, NP=1048576

```
h2 0.000107798791538840
h1 7.33764594586864e9
```

NE=12, NP=1048576

Modelo de tiempo para Función con 2 Niveles de Paralelismo en Memoria Compartida

Modelo: $t_{dos-niveles} = \frac{k_{s,2} \cdot NE \cdot NP}{h_1 \cdot h_2} + k_{h,1} \cdot h_1 + k_{h,2} \cdot h_2$

NE = 12
NP = 1048576

Constantes de Paralelismo:	Hilos		Tiempo
$k_{s,2}$ = 4,25793E-08	*h1*	*h2*	*teórico*
$k_{h,1}$ = 1,23081E-16	12	12	0,10
$k_{h,2}$ = 0,008377875	12	32	0,27
	12	36	0,30
Cambiando las Celdas	32	12	0,10
	32	32	0,27
	32	36	0,30
	36	12	0,10
	36	32	0,27
	36	36	0,30

Ilustración 32. NE=12, NP=1048576

```
h2 0.0000979417019938226
h1 6.66669377557459e9
```

NE=32, NP=1048576

Modelo de tiempo para Función con 2 Niveles de Paralelismo en Memoria Compartida

Modelo: $t_{dos-niveles} = \frac{k_{s,2} \cdot NE \cdot NP}{h_1 \cdot h_2} + k_{h,1} \cdot h_1 + k_{h,2} \cdot h_2$

NE = 20
NP = 1048576

Constantes de Paralelismo:	Hilos		Tiempo
$k_{s,2}$ = 4,25793E-08	*h1*	*h2*	*teórico*
$k_{h,1}$ = 1,23081E-16	12	12	0,11
$k_{h,2}$ = 0,008377875	12	32	0,27
	12	36	0,30
Cambiando las Celdas	32	12	0,10
	32	32	0,27
	32	36	0,30
	36	12	0,10
	36	32	0,27
	36	36	0,30

Ilustración 33. NE=32, NP=1048576

```
h2 0.000116122728017396
h1 7.90423948447561e9
```

NE=16, NP=1024

Modelo de tiempo para Función con 2 Niveles de Paralelismo en Memoria Compartida

Modelo: $t_{dos-niveles} = \frac{k_{s2} \cdot NE \cdot NP}{h_1 \cdot h_2} + k_{h,1} \cdot h_1 + k_{h,2} \cdot h_2$

NE = 32
NP = 1048576

Constantes de Paralelismo:	Hilos		Tiempo
k_{s2} = 4,25793E-08	*h1*	*h2*	*teórico*
$k_{h,1}$ = 1,23081E-16	12	12	0,11
$k_{h,2}$ = 0,008377875	12	32	0,27
	12	36	0,30
Cambiando las Celdas	32	12	0,10
	32	32	0,27
	32	36	0,30
	36	12	0,10
	36	32	0,27
	36	36	0,30

Ilustración 34. NE=16, NP=1024

```
h2 0.0000106949946931035
h1 7.27986680840130e8
```

NE=4, NP=1024

Modelo de tiempo para Función con 2 Niveles de Paralelismo en Memoria Compartida

Modelo: $t_{dos-niveles} = \frac{k_{s2} \cdot NE \cdot NP}{h_1 \cdot h_2} + k_{h,1} \cdot h_1 + k_{h,2} \cdot h_2$

NE = 4
NP = 1024

Constantes de Paralelismo:	Hilos		Tiempo
k_{s2} = 4,25793E-08	*h1*	*h2*	*teórico*
$k_{h,1}$ = 1,23081E-16	12	12	0,10
$k_{h,2}$ = 0,008377875	12	32	0,27
	12	36	0,30
Cambiando las Celdas	32	12	0,10
	32	32	0,27
	32	36	0,30
	36	12	0,10
	36	32	0,27
	36	36	0,30

Ilustración 35. NE=4, NP=1024

```
h2 6.73742447117750e-6
h1 4.58602871616790e8
```

NE=48, NP=1048576

Modelo de tiempo para Función con 2 Niveles de Paralelismo en Memoria Compartida

Modelo: $t_{dos-niveles} = \frac{k_{s2} \cdot NE \cdot NP}{h_1 \cdot h_2} + k_{h,1} \cdot h_1 + k_{h,2} \cdot h_2$

NE = 48
NP = 1024

Constantes de Paralelismo:	Hilos		Tiempo
k_{s2} = 4,25793E-08	*h1*	*h2*	*teórico*
$k_{h,1}$ = 1,23081E-16	12	12	0,10
$k_{h,2}$ = 0,008377875	12	32	0,27
	12	36	0,30
Cambiando las Celdas	32	12	0,10
	32	32	0,27
	32	36	0,30
	36	12	0,10
	36	32	0,27
	36	36	0,30

Ilustración 36. NE=48, NP=1048576

```
h2 0.00001542485150056 85
h1 1.04993847763119e9
```

NE=32, NP=1048576

Modelo de tiempo para Función con 2 Niveles de Paralelismo en Memoria Compartida

Modelo: $t_{dos-niveles} = \frac{k_{s2} \cdot NE \cdot NP}{h_1 \cdot h_2} + k_{h,1} \cdot h_1 + k_{h,2} \cdot h_2$

NE = 32
NP = 1048576

Constantes de Paralelismo:	Hilos		Tiempo
k_{s2} = 4,25793E-08	*h1*	*h2*	*teórico*
$k_{h,1}$ = 1,23081E-16	12	12	0,11
$k_{h,2}$ = 0,008377875	12	32	0,27
	12	36	0,30
Cambiando las Celdas	32	12	0,10
	32	32	0,27
	32	36	0,30
	36	12	0,10
	36	32	0,27
	36	36	0,30

Ilustración 37. NE=32, NP=1048576

```
h2 0.000135817966613014
h1 9.24485458387568e9
```

NE=32, NP=524288

Modelo de tiempo para Función con 2 Niveles de Paralelismo en Memoria Compartida

Modelo: $t_{dos-niveles} = \frac{k_{s2} \cdot NE \cdot NP}{h_1 \cdot h_2} + k_{h,1} \cdot h_1 + k_{h,2} \cdot h_2$

NE = 32
NP = 524288

Constantes de Paralelismo:	Hilos		Tiempo
k_{s2} = 4,25793E-08	*h1*	*h2*	*teórico*
$k_{h,1}$ = 1,23081E-16	12	12	0,11
$k_{h,2}$ = 0,008377875	12	32	0,27
	12	36	0,30
Cambiando las Celdas	32	12	0,10
	32	32	0,27
	32	36	0,30
	36	12	0,10
	36	32	0,27
	36	36	0,30

Ilustración 38. NE=32, NP=524288

```
h2 0.000107798791538840
h1 7.33764594586864e9
```

NE=16, NP=524288

Modelo de tiempo para Función con 2 Niveles de Paralelismo en Memoria Compartida

Modelo: $t_{dos-niveles} = \frac{k_{s2} \cdot NE \cdot NP}{h_1 \cdot h_2} + k_{h,1} \cdot h_1 + k_{h,2} \cdot h_2$

NE = 16
NP = 524288

Constantes de Paralelismo:	Hilos		Tiempo
k_{s2} = 4,25793E-08	*h1*	*h2*	*teórico*
$k_{h,1}$ = 1,23081E-16	12	12	0,10
$k_{h,2}$ = 0,008377875	12	32	0,27
	12	36	0,30
Cambiando las Celdas	32	12	0,10
	32	32	0,27
	32	36	0,30
	36	12	0,10
	36	32	0,27
	36	36	0,30

Ilustración 39. NE=16, NP=524288

```
h2 0.0000855599575448276
h1 5.82389344672104e9
```

5. Modelo de Tiempo en Memoria Distribuida

5.1. Análisis y obtención de constantes del sistema

Como se puede visualizar en la ilustración 43, nuestro modelo teórico ajusta correctamente con respecto a lo calculado experimentalmente. Sin embargo, podría parecer que el valor de la suma de la tabla de la función es demasiado grande, pero remarcar que esto es así puesto que cualquier mínima diferencia entre lo teórico y lo experimental es un número muy grande, puesto que estamos trabajando con valores de tiempos muy grandes.

Entrando más en detalles podemos observar cómo gracias al solver obtenemos dos gráficas muy similares, lo que significa que el ajuste es correcto. Para un hilo tenemos la ejecución más lenta ya que, no se puede explotar en absoluto la capacidad paralela del sistema, a medida que aumentamos el número de hilos podemos ver una mejora constante puesto que se paraleliza una mayor cantidad de computación con un coste de comunicación menor que la ganancia. Sin embargo, hemos comprobado experimentalmente que el rendimiento del programa empeora a partir de 18 procesos, ya que, de acuerdo a la configuración de nuestro algoritmo (N_GEN, T_POB, NGM, NEM, NPM), el coste de comunicación es mayor a la ganancia en cuanto a paralelización, debido a que la computación que realiza cada proceso no es lo suficientemente grande.

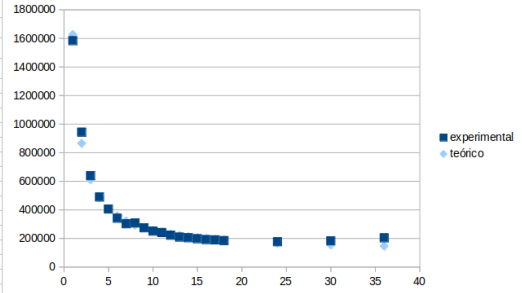
Modelo de tiempo para Memoria distribuida

Modelo: $t_{memoria-distribuida} = \frac{k_{cmp}}{p} + A * p3 + B * p2 + C * p + D$

Constantes de Paralelismo:
k_{cmp} = 1523679,145
A = 4,64741E-16
B = 1,69169E-14
C = 1,13646E-12
D = 102978,0951

Cambiando las Celdas

Procesos	Tiempo		Ajuste
p	*experimental*	*teórico*	$(t\_exp - t\_teo)^2$
1	1583702	1626657	1845152688,27403
2	943256	864817,67	6152571955,72614
3	639555	610871,14	822763619,038831
4	489758	483897,88	34340989,2566041
5	405632	407713,92	4334408,34719495
6	340966	356924,62	254677531,443984
7	302777	320646,54	319320619,054254
8	308268	293438	219929247,388154
9	274747	272275,78	6106938,49583362
10	250761	255346	21022313,5126221
11	241060	241494,38	188686,903343536
12	222283,0000	229951,36	58803702,5899838
13	208697	220184,18	131955378,326567
14	204271	211812,32	56871504,077564
15	197422	204556,7	50904012,6600922
16	191335	198208	47238702,2032911
17	189928	192606,28	7173184,48425652
18	184611	187626,94	9095873,11110316
24	176475	166464,73	100205582,148873
30	*183214*	153767,4	867102254,010962
36	*203579*	145302,52	3396148608,70643



Suma	10042451354,906800

Ilustración 40. Tabla con tiempos experimentales, constantes del sistema, tiempos teóricos y bondad de ajuste

5.2. Cálculo del tiempo de ejecución para diferentes valores de procesos y cálculo del número de procesos óptimo

NÚMERO DE PROCESOS	TIEMPO DE EJECUCIÓN (s)
2	864817,6700
6	356924,6200
10	255346,0000
14	211812,3200
18	187626,9400
22	172236,2381

24	166464,7300
26	161581,1391
28	157395,2074
30	153767,4000
32	150593,0684
34	147792,1876
36	145302,5200

Tabla 6. Cálculo de tiempos teóricos con distintos hilos

Para intentar calcular P_{opt} hemos intentado realizar el cálculo de manera analítica, pero los resultados obtenidos no fueron satisfactorios por lo que procedimos a intentarlo de manera iterativa.

El método que hemos intentado es mediante el uso de *Python*, mediante la prueba de diferentes valores desde 1 hasta un número muy elevado de manera que el resultado de la ecuación en valor absoluto nos de un número muy cercano a 0 (vid ilustración 44).

```
for x in range(1,1000000000000000):

    if abs(3*(4.647*10**-16)*x**4+2*(1.69*10**-14)*x**3+(1.1365*10**-12)*x**2-1523679.145)<0.01:
        print(x,3*(4.647*10**-16)*x**4+2*(1.69*10**-14)*x**3+(1.1365*10**-12)*x**2-1523679.145)
```

Ilustración 41. Extracto código Python para encontrar solución de forma iterativa

Hemos probado con las cotas de error 0.01, 0.1 y 10, pero en ningún caso obtuvimos una solución tras ejecutar el programa durante un minuto. Sin embargo, ejecutando con una cota de error de 100, obtuvimos las soluciones que se pueden observar en la ilustración 45 **¡Error! No se encuentra el origen de la referencia.**. Estas soluciones no son satisfactorias en absoluto, pero es todo lo que pudimos calcular.

```
181815 -80.11195037607104
181816 -46.59302625712007
181817 -13.073549083201215
181818 20.446481151739135
181819 53.967064453987405
181820 87.48820082936436
```

Ilustración 42. Soluciones obtenidas con método iterativo con una cota de error de 100