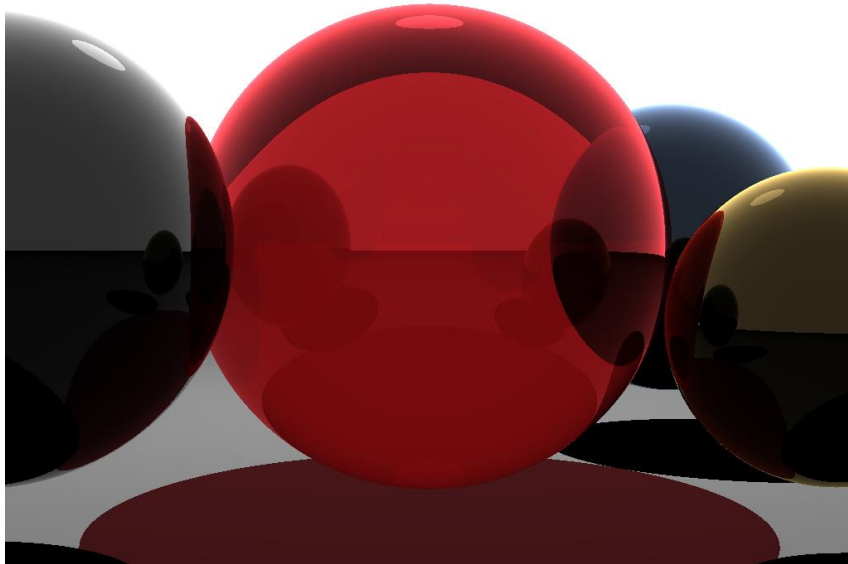


PRACTICA 2 TRAZADO DE RAYOS EN OPENMP

Ismael Fernández Herreruela



EJERCICIO 1

```
// Trace rays
for (y = 0; y < height; ++y) {
    ncalls_line = 0;

#pragma omp parallel for private(pixel,xx,yy,ncalls,raydir) reduction(+:ncalls_line)
    for (x = 0; x < width; ++x) {
        pixel = &image[y*width+x];
        xx = (2 * ((x + 0.5) * invWidth) - 1) * angle * aspectratio;
        yy = (1 - 2 * ((y + 0.5) * invHeight)) * angle;
        Vec3_new(&raydir, xx, yy, -1);
        Vec3_normalize(&raydir);
        ncalls = trace(pixel, &origin, &raydir, size, spheres, 0);
        ncalls_line += ncalls;
    }
    ncalls_line = ncalls_line/width;
    if (ncalls_line < min_ncalls_line)
        min_ncalls_line = ncalls_line;
    if (ncalls_line > max_ncalls_line)
        max_ncalls_line = ncalls_line;
    update_histogram(histo, ncalls_line);
    t2=omp_get_wtime();
}
```

En la paralelización de este apartado, hay que privatizar las variables ***pixel***, ***xx***, ***yy***, ***ncalls***, ***raydir*** ya que son las que se van a modificar, y por otro lado hay que hacer un reduction de la variable ***ncalls_line*** ya que tenemos un operador +=. En este caso no hay que hacer private la variable ***ncalls_line*** porque en este for, la variable ya esta inicializada.

EJERCICIO 2

```
// Trace rays
#pragma omp parallel for private(ncalls_line,x,xx,yy,ncalls,raydir,pixel)
for (y = 0; y < height; ++y) {
    ncalls_line = 0;
    for (x = 0; x < width; ++x) {
        pixel = &image[y*width+x];
        xx = (2 * ((x + 0.5) * invWidth) - 1) * angle * aspectratio;
        yy = (1 - 2 * ((y + 0.5) * invHeight)) * angle;
        Vec3_new(&raydir, xx, yy, -1);
        Vec3_normalize(&raydir);
        ncalls = trace(pixel, &origin, &raydir, size, spheres, 0);
        ncalls_line += ncalls;
    }
    ncalls_line = ncalls_line/width;

    if (ncalls_line < min_ncalls_line)
        min_ncalls_line = ncalls_line;
    if (ncalls_line > max_ncalls_line)
        max_ncalls_line = ncalls_line;
    update_histogram(histo, ncalls_line);
    t2=omp_get_wtime();
}
```

En este ejercicio teníamos que paralelizar el bucle externo, por lo que las cosas cambian un poco. Al estar en el bucle externo también entra la parte con los if. También podemos observar que ***ncalls_line*** se inicializa en este bucle por lo que no hay que ponerlo como reduction esta vez. El único cambio significativo en este apartado es el ya comentado, ***ncalls_line*** deja de ser reduction y se convierte en private, y también añadimos la variable ***x*** a private ya que se utiliza en el for interno.

EJERCICIO 3

RAYP1:

```
iferher@alumno.upv.es@kahan:~/p2$ cat slurm-17439.out
```

```
rayp1 static
```

```
Computing...
```

```
Complexity histogram:
```

Calls/pixel		Number of lines
[1.0 -- 2.5[194 *****
[2.5 -- 4.0[34 *****
[4.0 -- 5.5[143 *****
[5.5 -- 7.0[120 *****
[7.0 -- 8.5[198 *****
[8.5 -- 10.0[46 *****
[10.0 -- +inf[33 *****

```
Least complex line had 1.000 calls to trace per pixel.
```

```
Most complex line had 11.489 calls to trace per pixel.
```

```
Tiempo de ejecucion:10.161689
```

```
rayp1 static,1
```

```
Computing...
```

```
Complexity histogram:
```

Calls/pixel		Number of lines
[1.0 -- 2.5[194 *****
[2.5 -- 4.0[34 *****
[4.0 -- 5.5[143 *****
[5.5 -- 7.0[120 *****
[7.0 -- 8.5[198 *****
[8.5 -- 10.0[46 *****
[10.0 -- +inf[33 *****

```
Least complex line had 1.000 calls to trace per pixel.
```

```
Most complex line had 11.489 calls to trace per pixel.
```

```
Tiempo de ejecucion:4.810280
```

```
rayp1 dynamic
```

```
Computing...
```

```
Complexity histogram:
```

Calls/pixel		Number of lines
[1.0 -- 2.5[194 *****
[2.5 -- 4.0[34 *****
[4.0 -- 5.5[143 *****
[5.5 -- 7.0[120 *****
[7.0 -- 8.5[198 *****
[8.5 -- 10.0[46 *****
[10.0 -- +inf[33 *****

```
Least complex line had 1.000 calls to trace per pixel.
```

```
Most complex line had 11.489 calls to trace per pixel.
```

```
Tiempo de ejecucion:3.285955
```

En el caso de RAYP1 podemos observar que la opción que mejor resultado nos da es la de “Dynamic”, con 3,28s. En segundo lugar iría “static,1” con 4,81s. Y en ultimo lugar iría “static” con chunks por defecto con un resultado de 10,16s. “Dynamic” nos ofrece unos mejores resultados debido a que las iteraciones del bucle se distribuyen dinámicamente entre los hilos en tiempo de ejecución. En lugar de asignar un conjunto fijo de iteraciones a cada hilo, asigna a cada hilo un lote de iteraciones, y cuando un hilo termina su lote, recibe otro lote hasta que se completen todas las iteraciones. Por

otro lado, en “static” las iteraciones del bucle se dividen estáticamente entre los hilos antes de que comience la ejecución del bucle. Esto significa que cada hilo obtiene un número aproximadamente igual de iteraciones para procesar. La distribución es determinada en tiempo de compilación y se mantiene constante durante la ejecución del programa. Y en ultimo lugar “static,1” es similar a “static” pero garantiza que cada hilo reciba al menos una iteración para procesar.

RAYP2:

```
rayp2 static
Computing...
```

Complexity histogram:

Calls/pixel	Number of lines
[1.0 -- 2.5[194 *****
[2.5 -- 4.0[34 *****
[4.0 -- 5.5[143 *****
[5.5 -- 7.0[120 *****
[7.0 -- 8.5[198 *****
[8.5 -- 10.0[46 *****
[10.0 -- +inf[33 *****

Least complex line had 1.000 calls to trace per pixel.

Most complex line had 11.489 calls to trace per pixel.

Tiempo de ejecucion:5.882222

```
rayp2 static,1
```

```
Computing...
```

Complexity histogram:

Calls/pixel	Number of lines
[1.0 -- 2.5[194 *****
[2.5 -- 4.0[34 *****
[4.0 -- 5.5[143 *****
[5.5 -- 7.0[120 *****
[7.0 -- 8.5[198 *****
[8.5 -- 10.0[46 *****
[10.0 -- +inf[33 *****

Least complex line had 1.000 calls to trace per pixel.

Most complex line had 11.489 calls to trace per pixel.

Tiempo de ejecucion:3.485415

```
rayp2 dynamic
```

```
Computing...
```

Complexity histogram:

Calls/pixel	Number of lines
[1.0 -- 2.5[194 *****
[2.5 -- 4.0[34 *****
[4.0 -- 5.5[143 *****
[5.5 -- 7.0[120 *****
[7.0 -- 8.5[198 *****
[8.5 -- 10.0[46 *****
[10.0 -- +inf[33 *****

Least complex line had 1.000 calls to trace per pixel.

Most complex line had 11.489 calls to trace per pixel.

Tiempo de ejecucion:3.176196

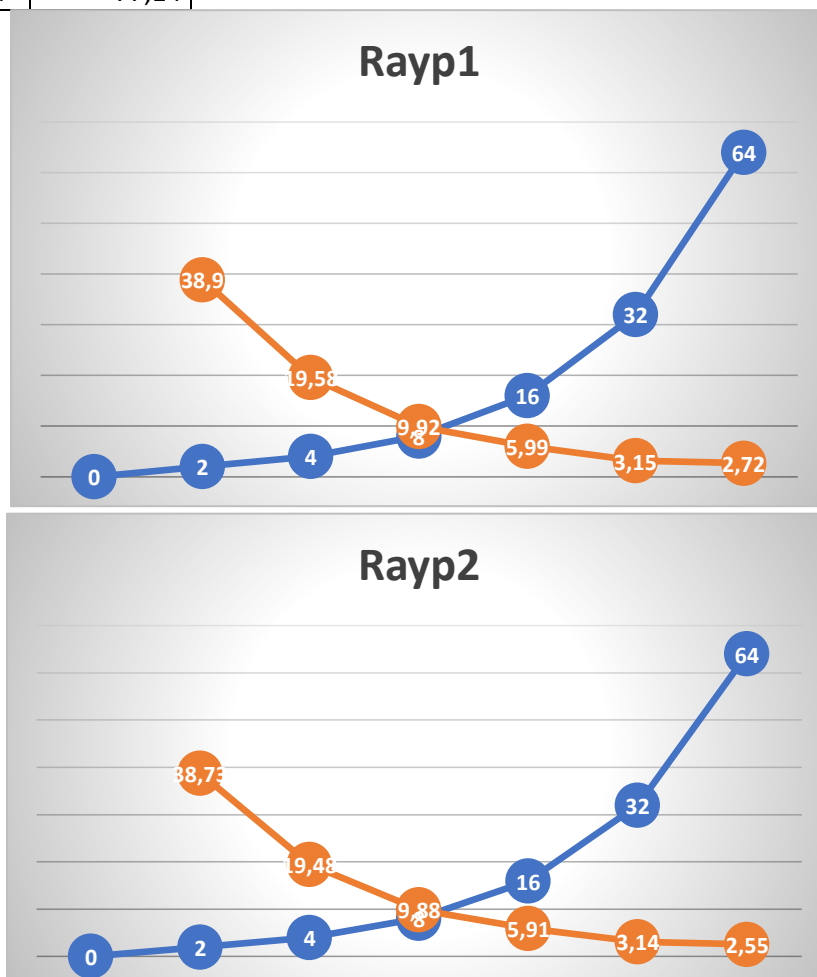
En el caso de RAYP2, la opción con “dynamic” también es la más rápida tardando 3,17s. En segundo lugar tendríamos a “static,1” con 3,48s y en último lugar “static” con 5,88s. La diferencia de rendimiento entre “dynamic” y “static” podría deberse a la distribución de carga desigual en la generación de rayos y en la llamada a la función trace() para calcular ncalls. Si la carga computacional varía significativamente entre diferentes regiones del espacio de píxeles, el uso de dynamic podría distribuir mejor la carga entre los hilos en comparación con static. Pero en el caso contrario podría hacer que fuese más lento, que es lo que ha ocurrido esta vez. Dependiendo del código, a veces será más óptimo utilizar “dynamic” y otras será más óptimo utilizar “static”.

EJERCICIO 4

Rayp1	2	4	8	16	32	64
	38,9	19,58	9,92	5,99	3,15	2,72

Rayp2	2	4	8	16	32	64
	38,73	19,48	9,88	5,91	3,14	2,55

Secuencial	77,14
------------	-------



SpeedUp p1	SpeedUp p2
1,98303342	1,99173767
3,93973442	3,95995893
7,77620968	7,80769231
12,8781302	13,0524535
24,4888889	24,566879
28,3602941	30,2509804

Eficiencia p1	Eficiencia p2
0,99151671	0,99586884
0,98493361	0,98998973
0,97202621	0,97596154
0,80488314	0,81577834
0,76527778	0,76771497
0,4431296	0,47267157

Obtenemos unos resultados muy similares en ambas opciones. Aunque la manera en que se maneja la variable `ncalls_line` entre los dos códigos difiere, el paralelismo es aplicado de manera similar en ambos casos. Es por eso que los resultados son bastante similares con unas diferencias ínfimas.

EJERCICIO 5

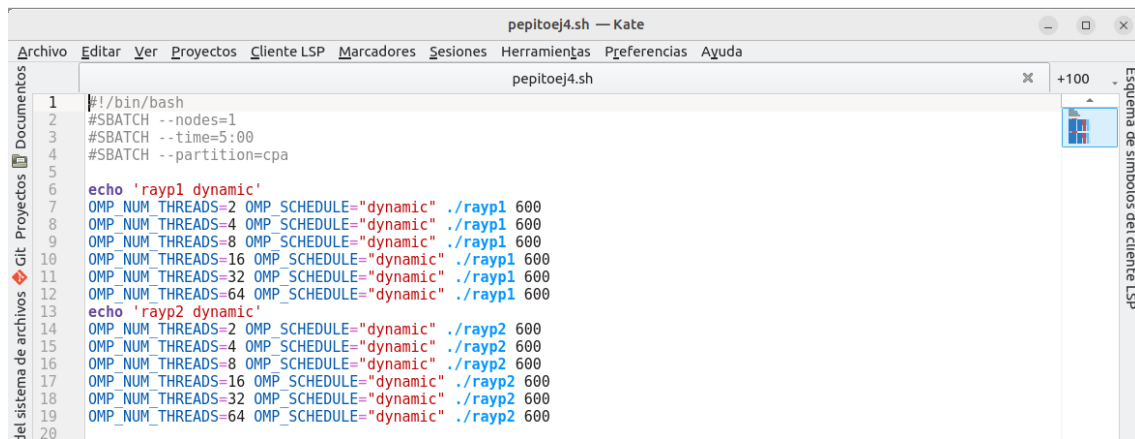
Para lanzar los programas en el kahan he utilizado un fichero `.sh` en el que dentro cambiaba las directrices a ejecutarse, como por ejemplo el numero de hilos o el tipo de Schedule.

```

1 #!/bin/bash
2 #SBATCH --nodes=1
3 #SBATCH --time=5:00
4 #SBATCH --partition=cpa
5 echo 'rayp1 static'
6 OMP_NUM_THREADS=32 OMP_SCHEDULE="static" ./rayp1 600
7 echo 'rayp1 static,1'
8 OMP_NUM_THREADS=32 OMP_SCHEDULE="static,1" ./rayp1 600
9 echo 'rayp1 dynamic'
10 OMP_NUM_THREADS=32 OMP_SCHEDULE="dynamic" ./rayp1 600
11 echo 'rayp2 static'
12 OMP_NUM_THREADS=32 OMP_SCHEDULE="static" ./rayp2 600
13 echo 'rayp2 static,1'
14 OMP_NUM_THREADS=32 OMP_SCHEDULE="static,1" ./rayp2 600
15 echo 'rayp2 dynamic'
16 OMP_NUM_THREADS=32 OMP_SCHEDULE="dynamic" ./rayp2 600
17

```

Por ejemplo en este caso, el fichero tiene 3 ejecuciones de `rayp1` y otras 3 de `rayp2`, cada una con un Schedule diferente. De esta manera nos ahorramos tiempo al ejecutarlas todas de una sola vez. También se puede observar que en todas las ejecuciones utilizamos 32 hilos.



```
1 #!/bin/bash
2 #SBATCH --nodes=1
3 #SBATCH --time=5:00
4 #SBATCH --partition=cpa
5
6 echo 'rayp1 dynamic'
7 OMP_NUM_THREADS=2 OMP_SCHEDULE="dynamic" ./rayp1 600
8 OMP_NUM_THREADS=4 OMP_SCHEDULE="dynamic" ./rayp1 600
9 OMP_NUM_THREADS=8 OMP_SCHEDULE="dynamic" ./rayp1 600
10 OMP_NUM_THREADS=16 OMP_SCHEDULE="dynamic" ./rayp1 600
11 OMP_NUM_THREADS=32 OMP_SCHEDULE="dynamic" ./rayp1 600
12 OMP_NUM_THREADS=64 OMP_SCHEDULE="dynamic" ./rayp1 600
13 echo 'rayp2 dynamic'
14 OMP_NUM_THREADS=2 OMP_SCHEDULE="dynamic" ./rayp2 600
15 OMP_NUM_THREADS=4 OMP_SCHEDULE="dynamic" ./rayp2 600
16 OMP_NUM_THREADS=8 OMP_SCHEDULE="dynamic" ./rayp2 600
17 OMP_NUM_THREADS=16 OMP_SCHEDULE="dynamic" ./rayp2 600
18 OMP_NUM_THREADS=32 OMP_SCHEDULE="dynamic" ./rayp2 600
19 OMP_NUM_THREADS=64 OMP_SCHEDULE="dynamic" ./rayp2 600
20
```

En este otro ejemplo, ejecutamos 6 veces rayp1 y rayp2, con el Schedule Dynamic pero cambiando en este caso el numero de hilos en cada ejecución, aumentandolos para así poder obtener el tiempo de ejecución de cada uno y hacer una comparativa y cálculos de SpeedUp y eficiencia.

Para lanzar estos archivos a ejecución en el kahan, utilizaba el siguiente comando:

```
sbatch ~/W/CPA/p2/material/nombre.sh
```

De esta manera añadíamos a la cola del kahan el archivo .sh a ejecutar. Posteriormente utilizaba el siguiente comando para visualizar los resultados:

```
cat slurm-****.out(donde los asteriscos son numeros)
```

Considero que el uso de 64 hilos es adecuado ya que en ciertas ocasiones, al utilizar muchos hilos puede acabar incrementándose el tiempo de ejecución(debido a la creación de todos esos hilos y la asignación de tareas). También, utilizando desde 2 a 64 hilos podemos obtener unos resultados suficientes para realizar comparaciones y sacar conclusiones.

EJERCICIO 6

Para agregar seguimiento del número de llamadas a la función trace por hilo y mostrar la línea más y menos compleja en términos de llamadas a trace por píxel, se realizaron los siguientes cambios:

- Se creó una estructura ThreadTraceData para almacenar el ID del hilo y el número de llamadas a trace por ese hilo.
- Se utilizó memoria dinámica para crear un arreglo de ThreadTraceData adaptado al número máximo de hilos posibles.
- Dentro de la región paralela, cada hilo actualiza su contador individual de llamadas a trace en la matriz calls_per_thread.
- Después de la región paralela, se iteró a través de la matriz calls_per_thread para imprimir el número de llamadas a trace por hilo.

-Durante la iteración de los píxeles, se calcularon las llamadas por píxel para cada línea, manteniendo un registro del mínimo y máximo de llamadas por píxel, junto con el número de línea respectivo.

-Finalmente, al final del programa, se imprimió la información sobre la línea más y menos compleja en términos de llamadas a trace por píxel.

La planificación de hilos en OpenMP puede tener un impacto en el número de llamadas realizadas por cada hilo. Si la planificación asigna diferentes tareas a diferentes hilos de manera desigual o altera el orden de ejecución de las tareas, puede influir en la cantidad total de llamadas a la función trace que realiza cada hilo. Por ejemplo, si se utiliza una planificación dynamic en lugar de static, donde las iteraciones del bucle se distribuyen de manera diferente entre los hilos, puede haber variaciones en la cantidad de trabajo realizado por cada hilo. En situaciones en las que una iteración del bucle requiere más o menos trabajo que otras, los hilos que manejan estas iteraciones pueden terminar con más o menos llamadas a trace respectivamente.