

Programación de procesadores multinúcleo usando OpenMP

Paula Poley Ceballos

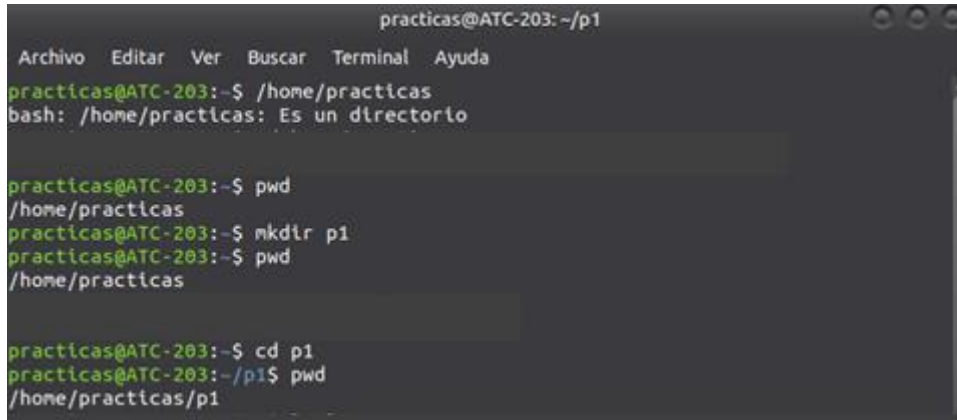
4º ISA, ASSB

Contenido

Pruebe a ejecutar varias veces éste último programa, estableciendo el número de hilos a crear mediante los dos mecanismos que hemos visto. Deduzca cuál de dichos mecanismos tiene preferencia.	7
Pruebe a compilar estos programas y ejecute varias veces éste último. Observe la numeración de los hilos y su orden de ejecución. ¿Qué lógica sigue el orden de ejecución de los hilos? ¿Es correcta la información devuelta por <code>omp_get_num_threads()</code> ? Modifique el programa para que muestre el valor correcto del número de hilos creados (a ser posible, una sola vez).	7
Pruebe a compilar estos programas y ejecutarlos, usando distintos valores para el número de hilos. ¿Realizan correctamente el cálculo de π ? Si no fuera así, ¿es capaz de identificar el problema o problemas de la versión paralela? ¿Cómo es posible que tarde más la versión paralela que la serie?	11
Modifique el programa anterior e incorpore <code>private(x)</code> en la directiva <code>#pragma</code> . Pruebe a compilar el programa y ejecutarlo. ¿Realiza correctamente el cálculo de π ? Si no fuera así, ¿es capaz de identificar más problemas?	12
¿Es capaz de explicar el problema de dicha sentencia en una ejecución paralela?	13
¿Es una solución al problema la modificación realizada? ¿Cuál es el rendimiento de esta solución?	13
¿Es correcta la modificación realizada? ¿Mejora el rendimiento esta solución? ¿Cuál es el motivo? ¿No bastaría con haber cambiado el <code>critical</code> por <code>atomic</code> , habida cuenta de que la operación es una de las admitidas como atómicas (<code>sum += 4...</code>), y así no necesitaríamos la variable auxiliar?	14
¿Funciona la modificación realizada? Compare el rendimiento de esta solución con el resto de las versiones del programa. Observe las mínimas modificaciones efectuadas sobre el código original (versión serie)	15

Para resolver todas las actividades indicadas en el apartado 3 de la guía de práctica 1 de la asignatura Arquitectura de Sistemas y Software de Base de 4º de Ingeniería de la Salud, antes voy a describir paso a paso lo que he hecho previamente.

Una vez encendido el ordenador del laboratorio lo inicio en Linux y abro la consola de comandos. Me voy al directorio prácticas y ahí creo una carpeta llamada “p1” en la que voy a guardar los archivos de texto. Con pwd se cuál es el directorio actual, en este caso: /home/practicas/p1. (Figura 1)



```
practicas@ATC-203: ~/p1
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
practicas@ATC-203:~$ /home/practicas
bash: /home/practicas: Es un directorio

practicas@ATC-203:~$ pwd
/home/practicas
practicas@ATC-203:~$ mkdir p1
practicas@ATC-203:~$ pwd
/home/practicas

practicas@ATC-203:~$ cd p1
practicas@ATC-203:~/p1$ pwd
/home/practicas/p1
```

(Figura 1)

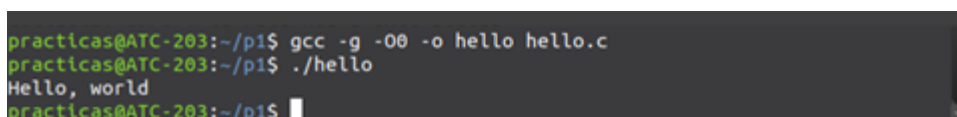
En la pantalla de inicio me voy a “Accesorios” y después a “Editor de texto”. De manera que ahí puedo escribir los programas y guardarlos en la carpeta que he creado anteriormente. Como podemos ver en la Figura 2 he escrito el programa llamado “hello.c”.



```
Abrir  hello.c  Guardar
~/p1
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char *argv[]) {
5     printf("Hello, world\n");
6     return 0;
7 }
```

(Figura 2)

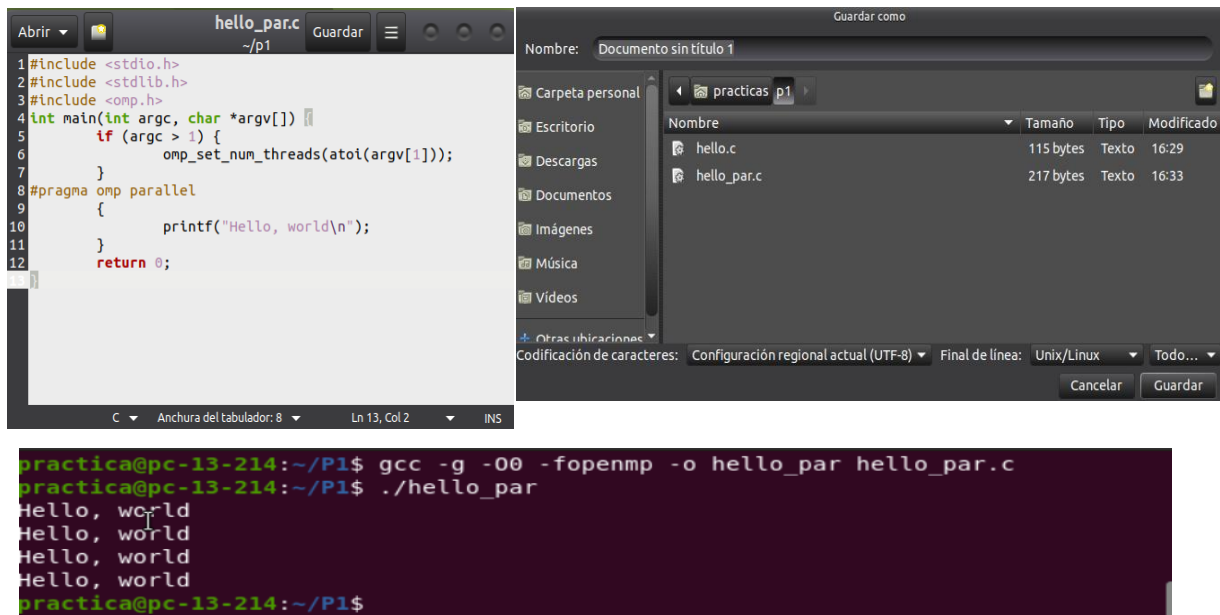
Vuelvo a la consola de comandos y escribo “gcc -g -O0 -o hello hello.c” para compilar el programa que antes he creado. Seguidamente como quiero ejecutar dicho programa escribo “. /nombredelprograma” en nuestro caso, “. /hello”. Al ejecutarlo me devuelve un solo “Hello, world” ya que así es como se ha configurado. Figura 3.



```
practicas@ATC-203:~/p1$ gcc -g -O0 -o hello hello.c
practicas@ATC-203:~/p1$ ./hello
Hello, world
practicas@ATC-203:~/p1$
```

(Figura 3)

Ahora lo que hago es paralelizar el programa, es decir ejecutar el programa varias veces de manera simultánea. Añado al programa anterior la cabecera `omp.h` e indico la parte del código que quiero que se ejecute en paralelo. La región paralela sería las líneas `...code....`. Como vemos en la Figura 4 guardo el programa, lo compilo escribiendo en la consola de comandos `gcc -g -O0 -fopenmp -o hello_par hello_par.c` a diferencia de la Figura 3 esta compilación añade `-fopenmp` para indicar que usamos OpenMP y seguidamente lo ejecuto `./hello_par`.



(Figura 4)

Para ver cuántos procesadores tiene el ordenador del laboratorio escribo en la consola de comandos `grep 'processor.*:' /proc/cpuinfo | wc -l` de manera que el programa creará tantos hilos como procesadores tenga disponibles. Como podemos ver en la Figura 5 el ordenador tiene 12 hilos y 6 procesadores porque el ordenador tiene tecnología Hyper-Threading (SMT), y entonces el número de procesadores es la mitad del ofrecido por el primero de los comandos. Figura 5

```

practicas@ATC-203:~/p1$ grep 'processor.*:' /proc/cpuinfo | wc -l
12

```

(Figura 5)

Con el comando “sudo lshw | less” muestra la información detallada sobre la configuración de hardware del sistema por pantalla. Figura 6

```
practicass@ATC-203:~/p1$ sudo lshw | less
[3]+  Detenido          sudo lshw | less
practicass@ATC-203:~/p1$
practicass@ATC-203:~/p1
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
*-pci
  descripción: Host bridge
  producto: 8th Gen Core Processor Host Bridge/DRAM Registers
  fabricante: Intel Corporation
  id físico: 100
  información del bus: pci@0000:00:00.0
  versión: 07
  anchura: 32 bits
  reloj: 33MHz
  configuración: driver=skl_uncore
  recursos: irq:0
*-pci:0
  descripción: PCI bridge
  producto: 6th-10th Gen Core Processor PCIe Controller (x16)
  fabricante: Intel Corporation
  id físico: 1
  información del bus: pci@0000:00:01.0
  versión: 07
  anchura: 32 bits
  reloj: 33MHz
  capacidades: pci pm msi pciexpress normal_decode bus_maste
r cap_list
  configuración: driver=pcieport
  recursos: irq:122 ioport:e000(size=4096) memoria:f6000000-
f70fffff ioport:e0000000(size=301989888)
```

(Figura 6)

Como se quiere establecer un número de hilos exactos paso el número de hilos que se quiere crear como argumento en la llamada con “omp_set_num_threads(int numThreads)” o también establezco el número de hilos deseado mediante una variable de entorno. Como podemos ver en la Figura 7 ejecuto el siguiente comando para que se creen 8 hilos “export OMP_NUM_THREADS=8”. Seguidamente ejecuto el programa con “./hello_par” y veo como imprime 8 veces “Hello, world”.

Si se nos olvida cuantos hilos habíamos puesto simplemente tenemos que poner el siguiente comando “echo \$OMP_NUM_THREADS” y nos lo dará.

```
practicass@ATC-203:~/p1$ export OMP_NUM_THREADS=8
practicass@ATC-203:~/p1$ ./hello_par
Hello, world
Hello, world
Hello, world
Hello, world
Hello, world
Hello, world
Hello, world
Hello, world
Hello, world
practicass@ATC-203:~/p1$ echo $OMP_NUM_THREADS
8
practicass@ATC-203:~/p1$
```

(Figura 7)

Ahora en vez de 8 hilos quiero 2 hilos y para ello lo que hago es borrar esta variable de entorno mediante “unset OMP_NUM_THREADS”, vuelvo a crear la variable “export OMP_NUM_THREADS=2” y ejecuto el programa “./hello_par”. Figura 8

Si analizamos la Figura 8 vemos como de principio a fin, lo que hago es: creo la variable de 8 hilos, ejecuto el programa con “echo”, veo cuantos hilos hay, borro la variable con “echo”, veo cuantos hilos hay (no sale ningún número porque claro, lo he borrado), vuelvo a crear la variable con 2 hilos, veo cuantos hilos son y ejecuto el programa.

```
practic@ATC-203:~/p1$ export OMP_NUM_THREADS=8
practic@ATC-203:~/p1$ ./hello_par
Hello, world
Hello, world
Hello, world
Hello, world
Hello, world
Hello, world
Hello, world
Hello, world
Hello, world
practic@ATC-203:~/p1$ echo $OMP_NUM_THREADS
8
practic@ATC-203:~/p1$ unset OMP_NUM_THREADS
practic@ATC-203:~/p1$ echo $OMP_NUM_THREADS

practic@ATC-203:~/p1$ export OMP_NUM_THREADS=2
practic@ATC-203:~/p1$ echo $OMP_NUM_THREADS
2
practic@ATC-203:~/p1$ ./hello_par
Hello, world
Hello, world
practic@ATC-203:~/p1$
```

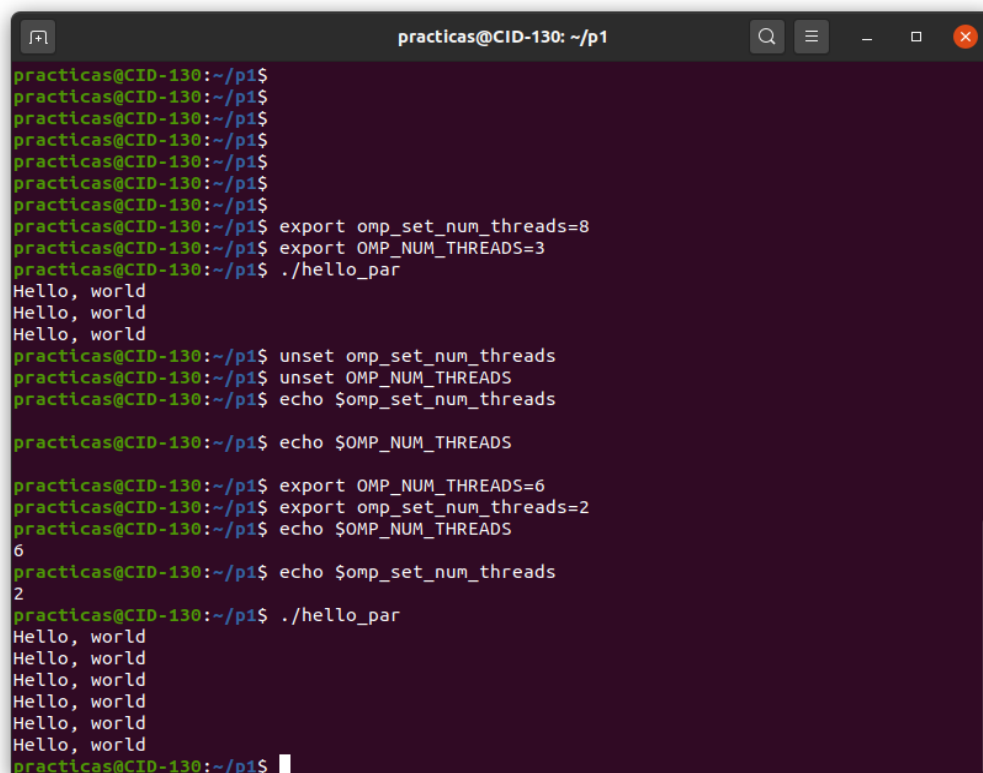
(Figura 8)

En conclusión, veo que cuando le indico que quiero 8 hilos, el programa me está lanzando 8 copias diferentes que corren de manera independiente y pueden correr en CPUs distintas. Si no hay suficientes CPUs entonces corren juntas las copias, de mi código con el export omp_num_threads que lo hago desde la shell. A ser posible cada hilo corre en CPUs distintas, sino las reparte. Cada hilo es un proceso diferente que corresponde al mismo código fuente.

Cada copia se diferencia en que cada hilo, el sistema operativo con las librerías le asigna un identificador de hilo único que es distinto con respecto a los demás hilos. La numeración va desde 0 (el hilo 'principal') hasta numThreads-1.

Pruebe a ejecutar varias veces éste último programa, estableciendo el número de hilos a crear mediante los dos mecanismos que hemos visto. Deduzca cuál de dichos mecanismos tiene preferencia.

Respondiendo a la pregunta, por preferencia primero sería la variable de entorno OMP_NUM_THREADS. Ya que si por ejemplo creo OMP_NUM_THREADS con un número de hilos y después también con la primitiva omp_set_num_threads(int numThreads) creo otro número de hilos, al ejecutar el programa (./hello_par) saldrá el número de hilos que le haya puesto en la variable de entorno ya que tiene preferencia. Esto lo podemos ver en la figura a , en la que da igual el orden que pongo para la variable de entorno para establecer el número exacto de hilos a crear que siempre cuando se ejecuta el programa solo tiene en cuenta a esta y no a la primitiva.

A terminal window titled 'practicass@CID-130: ~/p1' with a dark purple background. It shows a series of commands and their outputs. First, several empty prompts are shown. Then, 'export omp_set_num_threads=8' and 'export OMP_NUM_THREADS=3' are entered. The command './hello_par' is run, resulting in three 'Hello, world' outputs. Next, 'unset omp_set_num_threads' and 'unset OMP_NUM_THREADS' are entered, followed by 'echo \$omp_set_num_threads' which outputs an empty line. Then 'echo \$OMP_NUM_THREADS' outputs '6'. After 'export OMP_NUM_THREADS=6' and 'export omp_set_num_threads=2', 'echo \$OMP_NUM_THREADS' outputs '6' and 'echo \$omp_set_num_threads' outputs '2'. Finally, './hello_par' is run again, resulting in six 'Hello, world' outputs.

```
practicass@CID-130:~/p1$  
practicass@CID-130:~/p1$  
practicass@CID-130:~/p1$  
practicass@CID-130:~/p1$  
practicass@CID-130:~/p1$  
practicass@CID-130:~/p1$  
practicass@CID-130:~/p1$  
practicass@CID-130:~/p1$ export omp_set_num_threads=8  
practicass@CID-130:~/p1$ export OMP_NUM_THREADS=3  
practicass@CID-130:~/p1$ ./hello_par  
Hello, world  
Hello, world  
Hello, world  
practicass@CID-130:~/p1$ unset omp_set_num_threads  
practicass@CID-130:~/p1$ unset OMP_NUM_THREADS  
practicass@CID-130:~/p1$ echo $omp_set_num_threads  
  
practicass@CID-130:~/p1$ echo $OMP_NUM_THREADS  
6  
practicass@CID-130:~/p1$ export OMP_NUM_THREADS=6  
practicass@CID-130:~/p1$ export omp_set_num_threads=2  
practicass@CID-130:~/p1$ echo $OMP_NUM_THREADS  
6  
practicass@CID-130:~/p1$ echo $omp_set_num_threads  
2  
practicass@CID-130:~/p1$ ./hello_par  
Hello, world  
Hello, world  
Hello, world  
Hello, world  
Hello, world  
Hello, world  
practicass@CID-130:~/p1$
```

Figura a.

Pruebe a compilar estos programas y ejecute varias veces éste último. Observe la numeración de los hilos y su orden de ejecución. ¿Qué lógica sigue el orden de ejecución de los hilos? ¿Es correcta la información devuelta por omp_get_num_threads()? Modifique el programa para que muestre el valor correcto del número de hilos creados (a ser posible, una sola vez).

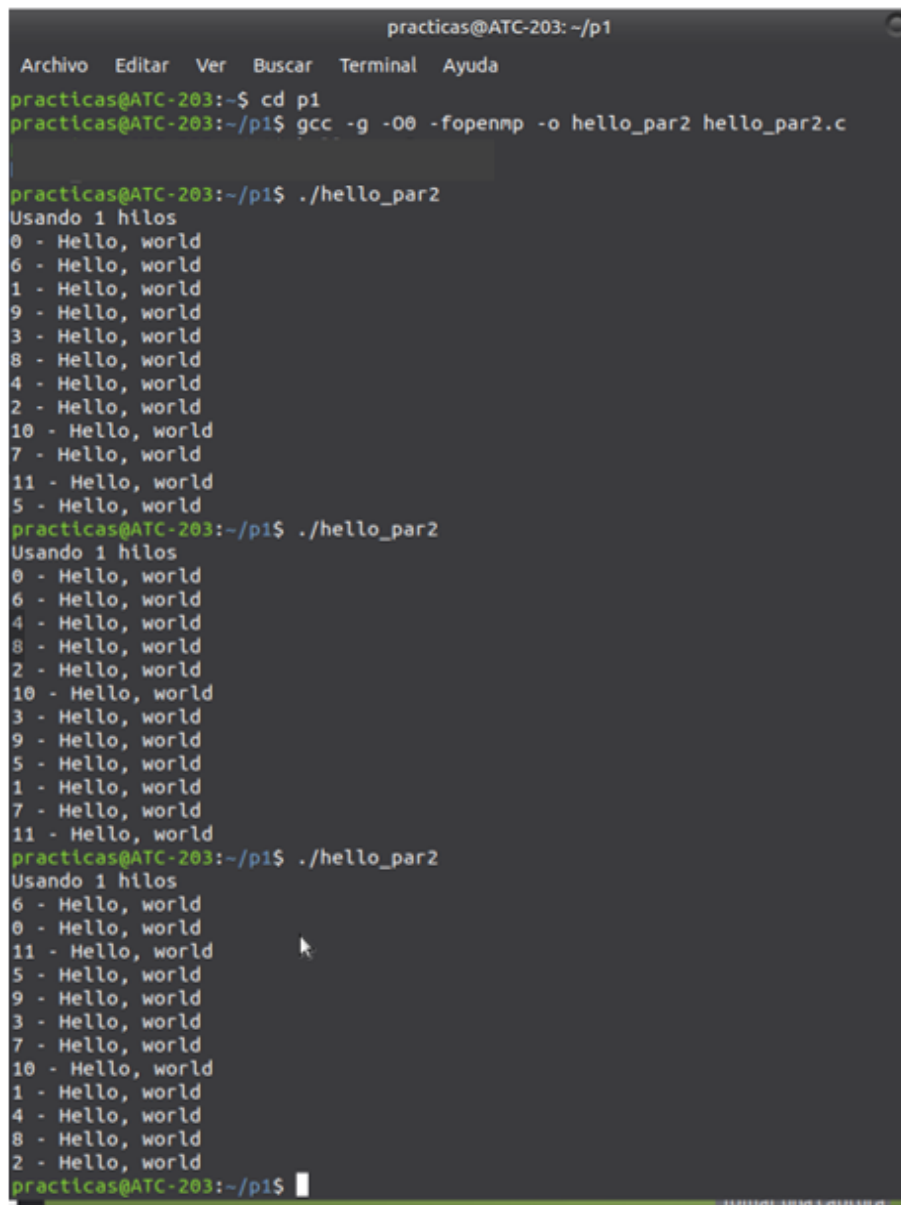
A continuación, veremos los identificadores de los hilos.

Copio el programa y lo guardo como “hello_par2.c”



```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <omp.h>
4 int main(int argc, char *argv[]) {
5     if (argc > 1) {
6         omp_set_num_threads(atoi(argv[1]));
7     }
8     printf("Usando %d hilos\n", omp_get_num_threads());
9     #pragma omp parallel
10 {
11     printf("%d - Hello, world\n", omp_get_thread_num());
12 }
13 return 0;
14 }
```

Compilo el programa como ya he hecho anteriormente y lo ejecuto. Vemos en la Figura 10 que estos identificadores únicos de cada hilo se sitúan en el lado izquierdo al principio de cada hilo. También podemos apreciar como hemos ejecutado 3 veces el programa y en cada uno los identificadores cambian, comprobando que el orden de los hilos no sigue ninguna lógica .

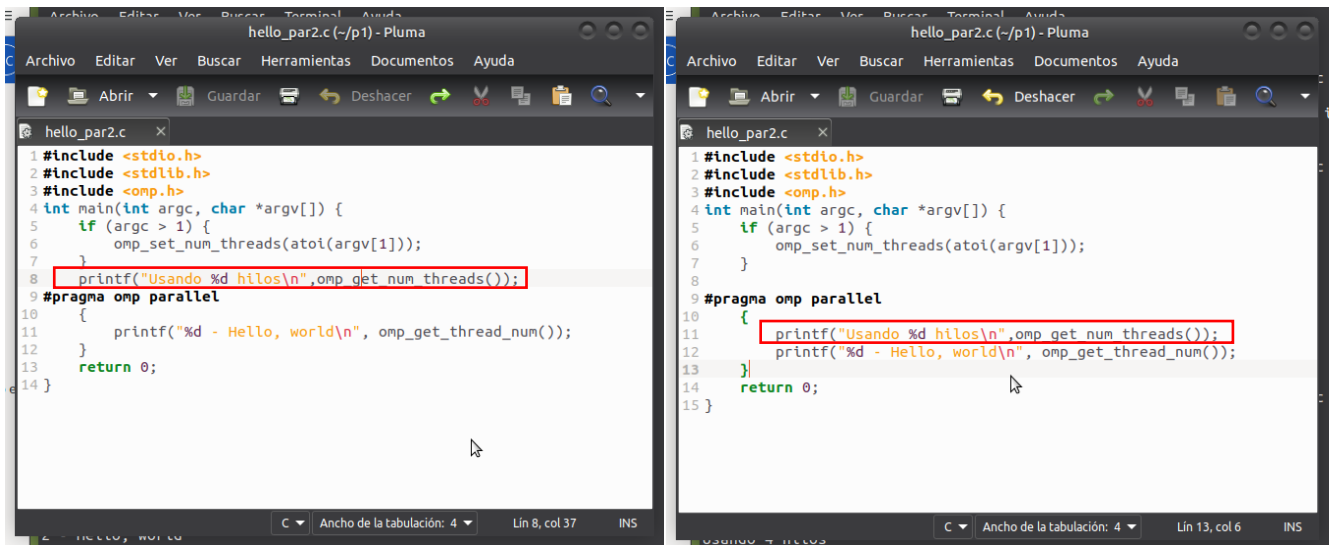


```
practicass@ATC-203: ~/p1
Archivo Editar Ver Buscar Terminal Ayuda
practicass@ATC-203:~$ cd p1
practicass@ATC-203:~/p1$ gcc -g -O0 -fopenmp -o hello_par2 hello_par2.c
practicass@ATC-203:~/p1$ ./hello_par2
Usando 1 hilos
0 - Hello, world
6 - Hello, world
1 - Hello, world
9 - Hello, world
3 - Hello, world
8 - Hello, world
4 - Hello, world
2 - Hello, world
10 - Hello, world
7 - Hello, world
11 - Hello, world
5 - Hello, world
practicass@ATC-203:~/p1$ ./hello_par2
Usando 1 hilos
0 - Hello, world
6 - Hello, world
4 - Hello, world
8 - Hello, world
2 - Hello, world
10 - Hello, world
3 - Hello, world
9 - Hello, world
5 - Hello, world
1 - Hello, world
7 - Hello, world
11 - Hello, world
practicass@ATC-203:~/p1$ ./hello_par2
Usando 1 hilos
6 - Hello, world
0 - Hello, world
11 - Hello, world
5 - Hello, world
9 - Hello, world
3 - Hello, world
7 - Hello, world
10 - Hello, world
1 - Hello, world
4 - Hello, world
8 - Hello, world
2 - Hello, world
practicass@ATC-203:~/p1$
```

(Figura 9)

La información devuelta por `omp_get_num_threads()` no es correcta ya que no muestra el valor correcto del número de hilos creados como podemos ver en la Figura 9. Ya que sale que “Usando 1 hilos” pero en realidad no es así, se muestran 12 hilos.

Esto ocurre porque el código no ha entrado en la región paralela del programa. Para solucionarlo cambio el código como podemos ver en la Figura 10. Lo que hago es cortar el print de “Usando ...” y lo pego dentro de la región paralela. (`#pragma omp parallel`) de manera que ahora al ejecutar el programa me dará el número de hilos correcto.



(Figura 10)

Anteriormente he creado la variable de entorno para crear 4 hilos, he compilado el programa y lo he ejecutado. Ahora se compila el nuevo programa con el cambio hecho (Figura 10) y ejecuto el programa. Vemos como pasa de poner por pantalla “Usando 1 hilos” a “Usando 4 hilos”



(Figura 11)

Pero ¿Qué sigue ocurriendo? que el texto se repite cada vez, es decir uno por cada hilo que imprima. Entonces para que lo ponga solo una vez me puedo agarrar a su identificador de hilo, de manera que pongo un `if` y que el primero de los hilos ponga su

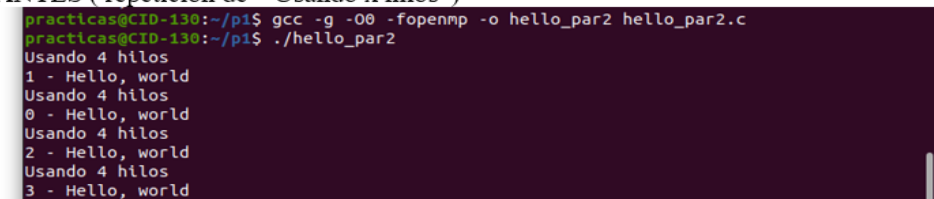
identificador, si mi identificador de hilo es 0 hazme x cosa. A continuación, lo vemos en la figura b en un cuadrado amarillo.



Figura b

De manera que vemos una gran diferencia cuando compilo y ejecuto el programa. Figura c. Solo se ve una fila de texto en la que se diga “Usando x hilos”.

ANTES (repetición de “Usando x hilos”)



DESPUÉS (Solucionado)

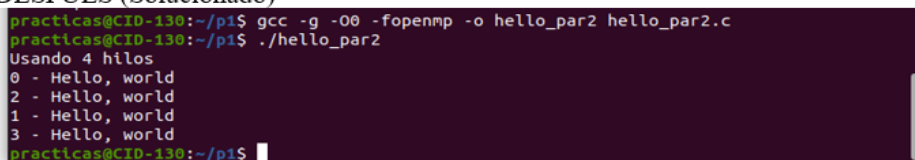
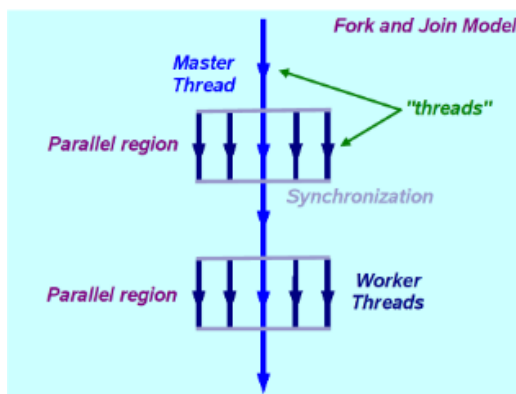


Figura c

Lo que ocurre está bien esquematizado en la siguiente imagen.



Paralelismo. Modelo Fork & Join

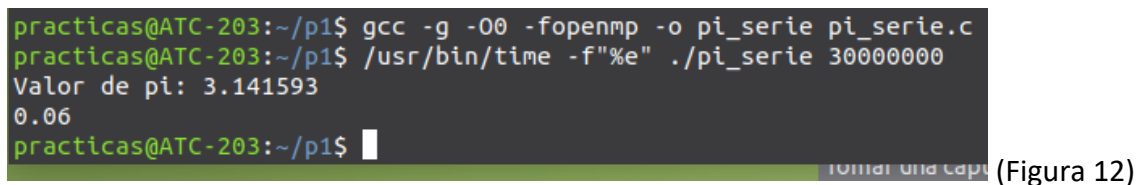
Pruebe a compilar estos programas y ejecutarlos, usando distintos valores para el número de hilos. ¿Realizan correctamente el cálculo de π ? Si no fuera así, ¿es capaz de identificar el problema o problemas de la versión paralela? ¿Cómo es posible que tarde más la versión paralela que la serie?

Creo un nuevo archivo de texto llamado “pi_serie.c” con el siguiente programa en versión no paralela.



```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(int argc, char *argv[]) {
4     int i;
5     double x, pi, sum = 0.0;
6     long numSteps = atol(argv[1]);
7     double step = 1.0 / (double)numSteps;
8
9     for (i=0; i<numSteps; ++i) {
10         x = (i+0.5)*step;
11         sum += 4.0/(1.0+x*x);
12     }
13     pi = step * sum;
14
15     printf("Valor de pi: %f\n", pi);
16     return 0;
17 }
```

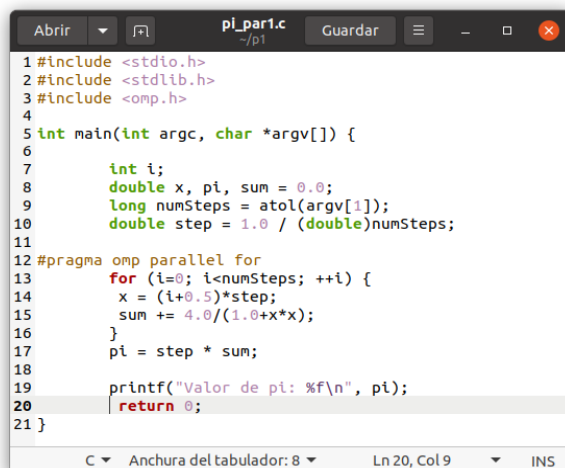
Primeramente, compilo el programa y lo ejecuto con “/usr/bin/time -f”%e” ./pi_serie 30000000” para que calcule el tiempo en segundos que tarda en ejecutarse dicho programa. Como podemos ver en la Figura 12 , tardará 0,06 segundos y da el valor correcto de pi.



```
practicass@ATC-203:~/p1$ gcc -g -O0 -fopenmp -o pi_serie pi_serie.c
practicass@ATC-203:~/p1$ /usr/bin/time -f"%e" ./pi_serie 30000000
Valor de pi: 3.141593
0.06
practicass@ATC-203:~/p1$
```

(Figura 12)

Veamos ahora lo que ocurre con el valor de pi con un nuevo programa, pero esta vez con la versión paralela. Creo un nuevo archivo de texto llamado “pi_par1.c” , lo guardo, compilo y lo ejecuto.



```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <omp.h>
4
5 int main(int argc, char *argv[]) {
6
7     int i;
8     double x, pi, sum = 0.0;
9     long numSteps = atol(argv[1]);
10    double step = 1.0 / (double)numSteps;
11
12    #pragma omp parallel for
13    for (i=0; i<numSteps; ++i) {
14        x = (i+0.5)*step;
15        sum += 4.0/(1.0+x*x);
16    }
17    pi = step * sum;
18
19    printf("Valor de pi: %f\n", pi);
20    return 0;
21 }
```

Vemos en la Figura 13 que el tiempo que tarda en ejecutarse el programa es mayor que en la versión no paralela (Figura 12) y además el valor de pi no es correcto dándonos 0.691362.

```
practicass@ATC-203:~/p1$ gcc -g -O0 -fopenmp -o pi_par1 pi_par1.c
practicass@ATC-203:~/p1$ /usr/bin/time -f"%e" ./pi_par1 30000000
Valor de pi: 0.691362
0.12
practicass@ATC-203:~/p1$
```

(Figura 13)

Aunque cambie continuamente el número de hilos que quiero crear, en la versión paralela del programa el valor de pi sigue siendo incorrecto y los valores de tiempo que tarda en ejecutarse también varían. En cambio, aunque cambie el número de hilos en la versión no paralela del programa el valor de pi no cambia. Figura 14

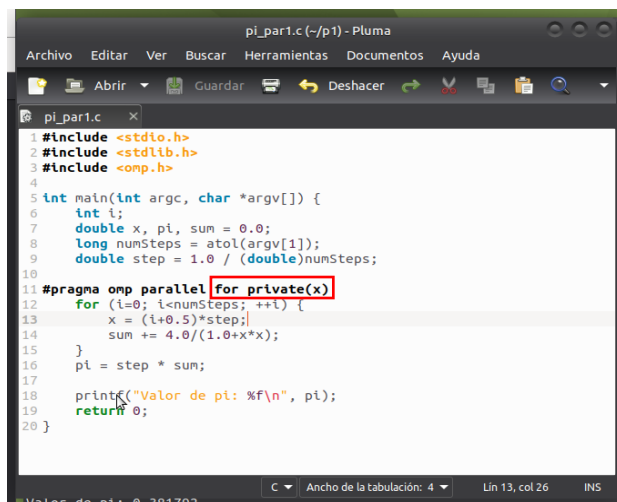
```
practicass@ATC-203:~/p1$ export OMP_NUM_THREADS=20
practicass@ATC-203:~/p1$ /usr/bin/time -f"%e" ./pi_serie 30000000
Valor de pi: 3.141593
0.06
practicass@ATC-203:~/p1$ gcc -g -O0 -fopenmp -o pi_par1 pi_par1.c
practicass@ATC-203:~/p1$ /usr/bin/time -f"%e" ./pi_par1 30000000
Valor de pi: 0.259290
0.13
practicass@ATC-203:~/p1$ export OMP_NUM_THREADS=10
practicass@ATC-203:~/p1$ /usr/bin/time -f"%e" ./pi_par1 30000000
Valor de pi: 0.344838
0.12
practicass@ATC-203:~/p1$
```

(Figura 14)

El problema de la versión paralela del programa es que la variable x en el interior del bucle inicializa con un valor y en la siguiente sentencia se usa, pero no hay garantía ninguna de que otro hilo no modifique x entre ambas sentencias alterando los cálculos. Para esto hay dos soluciones o pongo private(x) en la directiva o pongo privada la variable x, declarándola como double en el interior del bucle.

Modifique el programa anterior e incorpore private(x) en la directiva #pragma. Pruebe a compilar el programa y ejecutarlo. ¿Realiza correctamente el cálculo de pi? Si no fuera así, ¿es capaz de identificar más problemas?

Yo he elegido poner private(x) como podemos verlo en la Figura 15 con un rectángulo rojo.



```
pi_par1.c (-/p1) - Pluma
Archivo  Editar  Ver  Buscar  Herramientas  Documentos  Ayuda
Abrir  Guardar  Deshacer
pi_par1.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <omp.h>
4
5 int main(int argc, char *argv[]) {
6     int i;
7     double x, pi, sum = 0.0;
8     long numSteps = atol(argv[1]);
9     double step = 1.0 / (double)numSteps;
10
11     #pragma omp parallel for private(x)
12     for (i=0; i<numSteps; ++i) {
13         x = (i+0.5)*step;
14         sum += 4.0/(1.0+x*x);
15     }
16     pi = step * sum;
17
18     printf("Valor de pi: %f\n", pi);
19     return 0;
20 }
```

Valor de pi: 0.381792

(Figura 15)

Vuelvo a compilar el programa y lo ejecuto. Vemos como sale igual, el mismo error, no calcula bien el valor de pi. Figura 16

```
practic@ATC-203:~/p1$ gcc -g -O0 -fopenmp -o pi_par1 pi_par1.c
practic@ATC-203:~/p1$ /usr/bin/time -f"%e" ./pi_par1 30000000
Valor de pi: 0.480581
0.09
practic@ATC-203:~/p1$
```

(Figura 16)

El problema del mal cálculo de pi está en la ecuación “ $\text{sum} += 4.0/(1.0+x*x)$ ” en el código del programa.

¿Es capaz de explicar el problema de dicha sentencia en una ejecución paralela?

Pasa lo mismo que he comentado antes, aunque es una sola sentencia, varios hilos modifican simultáneamente el valor de la variable sum. Quien da una solución ante esto es OpenMP, en el que se puede definir una región crítica, estas se ejecutan secuencialmente, rompiendo el paralelismo en esa zona y proporcionando un acceso seguro a memoria compartida.

¿Es una solución al problema la modificación realizada? ¿Cuál es el rendimiento de esta solución?

Guardamos el programa llamándolo “pi_par3.c”



```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <omp.h>
4 int main(int argc, char *argv[]) {
5     int i;
6     double x, pi, sum = 0.0;
7     long numSteps = atol(argv[1]);
8     double step = 1.0 / (double)numSteps;
9     #pragma omp parallel for private(x)
10    for (i=0; i<numSteps; ++i) {
11        x = (i+0.5)*step;
12        #pragma omp critical
13        sum += 4.0/(1.0+x*x);
14    }
15    pi = step * sum;
16    printf("Valor de pi: %f\n", pi);
17    return 0;
18 }
```

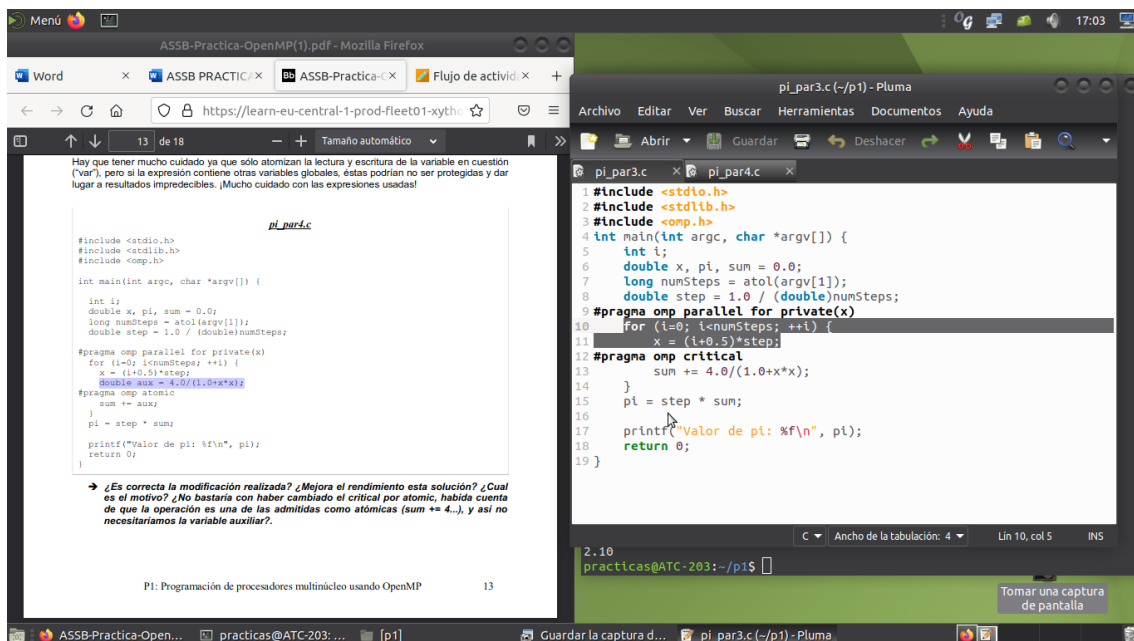
Compilo el programa y lo ejecuto. Vemos en la Figura 17 que ahora si nos da el valor correcto de pi, siendo entonces esta una posible solución. Pero lo que ocurre con respecto a las demás es que el rendimiento es peor con diferencia. En este programa, el tiempo que tarda en ejecutarlo es de 2.37 segundos.

```
practicass@ATC-203: ~/p1
Archivo Editar Ver Buscar Terminal Ayuda
practicass@ATC-203:~/p1$ gcc -g -O0 -fopenmp -o pi_par3 pi_par3.c
practicass@ATC-203:~/p1$ /usr/bin/time -f"%e" ./pi_par3 30000000
Valor de pi: 3.141593
2.37
practicass@ATC-203:~/p1$
```

(Figura 17)

¿Es correcta la modificación realizada? ¿Mejora el rendimiento esta solución? ¿Cuál es el motivo? ¿No bastaría con haber cambiado el critical por atomic, habida cuenta de que la operación es una de las admitidas como atómicas (sum += 4...), y así no necesitaríamos la variable auxiliar?

Lo que cambia pi_par4.c es lo señalado en la figura 18 , las atómicas (atomic) mejoran mucho el rendimiento, pero están definidas para operaciones muy sencillas, entonces para una expresión no tan simple no tiene poque ejecutar tan rápido , para expresiones sencillas si.



(Figura 18)

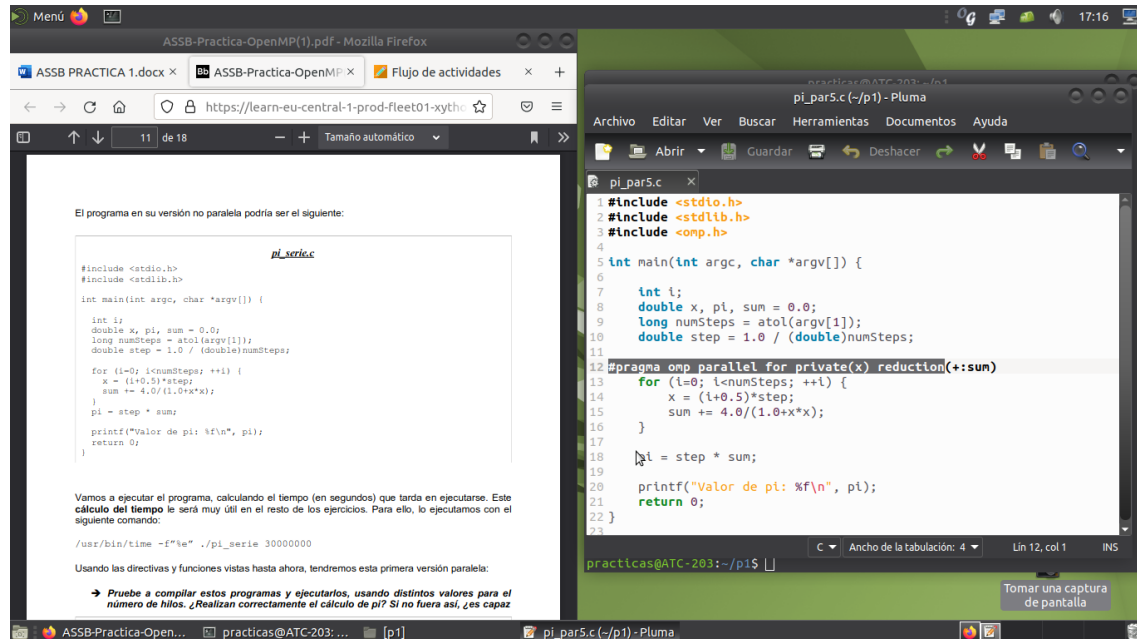
Ahora voy a guardar otro programa llamado “pi_par4.c” el cuál ha sufrido una modificación, lo compilo y ejecuto. De manera que se ve que mejora el rendimiento porque tarda menos tiempo, ya que hace más rápido los procesos.

```
practicass@ATC-203: ~/p1
Archivo Editar Ver Buscar Terminal Ayuda
practicass@ATC-203:~/p1$ gcc -g -O0 -fopenmp -o pi_par4 pi_par4.c
practicass@ATC-203:~/p1$ /usr/bin/time -f"%e" ./pi_par4 30000000
Valor de pi: 3.141593
2.34
practicass@ATC-203:~/p1$
```

(Figura 18)

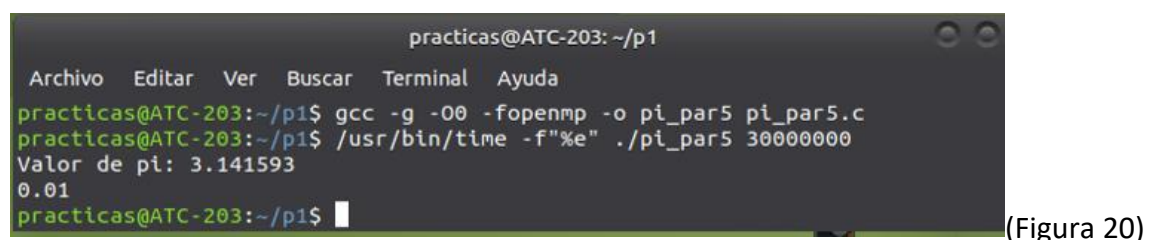
¿Funciona la modificación realizada? Compare el rendimiento de esta solución con el resto de las versiones del programa. Observe las mínimas modificaciones efectuadas sobre el código original (versión serie)

Lo que cambia del original (serie) a la version 5 (pi_par5) es lo que he subrayado en gris. Se ha paralelizado el programa ofreciendo reducciones para evitar el uso de mecanismos de sincronización entre hilos que es más pesado.



Si lo compilo y ejecuto el valor de pi es el correcto y además el tiempo que tarda en ejecutarse el programa es muy pequeño, en comparación con los demás vistos este sería el menor. Por lo tanto, sí que la modificación hecha funciona correctamente.

Figura 19



(Figura 20)

A continuación, en la Figura 21 vamos a ver todos los programas vistos anteriormente con el valor de pi y el tiempo que tarda en ejecutar el programa. Es decir, pi_serie, pi_par1, pi_par3, pi_par4 y pi_par5.

```
practicass@ATC-203: ~/p1
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
practicass@ATC-203:~/p1$ /usr/bin/time -f"%e" ./pi_serie 30000000
Valor de pi: 3.141593
0.06
practicass@ATC-203:~/p1$ /usr/bin/time -f"%e" ./pi_par1 30000000
Valor de pi: 0.515963
0.09
practicass@ATC-203:~/p1$ /usr/bin/time -f"%e" ./pi_par3 30000000
Valor de pi: 3.141593
3.59
practicass@ATC-203:~/p1$ /usr/bin/time -f"%e" ./pi_par4 30000000
Valor de pi: 3.141593
2.27
practicass@ATC-203:~/p1$ /usr/bin/time -f"%e" ./pi_par5 30000000
Valor de pi: 3.141593
0.01
practicass@ATC-203:~/p1$
```

(Figura 21)

Vemos que el valor de pi es correcto en las versiones pi_serie, pi_par3,4,5 menos en pi_par1. Y que la versión del programa que mejor rendimiento obtiene es pi_par5 (paralelismo privado con reducciones), seguido de pi_par1. Siendo el programa con peor rendimiento pi_par3.