EXPLOTAR EL POTENCIAL DE LAS ARQUITECTURAS MODERNAS

Práctica 4 Arquitectura de Ordenadores

Descripción breve

Los ejercicios realizados están enfocados a familiarizarse con la herramienta de programación OpenMP y empezar a entender un nuevo tipo de arquitectura como son las arquitecturas *multicore*

Adrián San Felipe Martín y Carlos Miret Fiuza

Ejercicio 0: Información sobre la topología del sistema

Para llevar a cabo la realización de este ejercicio, nos hemos conectado al cluster de la universidad para poder determinar los datos de la **cpuinfo** (adjuntados en la entrega). Se trata de un equipo AMD con **800 MHz** de frecuencia y tamaño de caché de **512Kb.** Tiene 8 cores (CPU´s físicas) y 8 cores (CPU´s virtuales) denominados siblings, por lo que determinamos que no existe *HyperThreading* al no existir el doble de virtuales que de físicos. (ver **cpuinfo.txt** adjuntado en la entrega).

Ejercicio 1: Programas básicos de OpenMP

1.1 ¿Se pueden lanzar más threads que cores tenga el sistema? ¿Tiene sentido hacerlo?

Se puede, pero carece de sentido ya que en cuanto a rendimiento se refiere, no se obtendría mejoría. Esto se debe a que cada core se encarga de la ejecución de un hilo, por lo que no se podría obtener mejora a la hora de tener más hilos que cores.

1.2 ¿Cuántos threads debería utilizar en los ordenadores del laboratorio? ¿y en el clúster? ¿y en su propio equipo?

Se debería utilizar tantos threads como cores haya, en el caso del cluster 16 y por último, en el caso de mi equipo 1.

1.3 Modifique el programa omp1.c para utilizar las tres formas de elegir el número de threads y deduzca la prioridad entre ellas

(Ver modificaciones en omp1.c). La variable entorno OMP_NUM_THREADS tiene menos prioridad que la función propuesta y esta (al intentar diferentes formas de ejecución vemos que la función predomina frente a la variable de entorno), a su vez tiene menos prioridad que la cláusula dentro de la región paralela.

1.4 ¿Cómo se comporta OpenMP cuando declaramos una variable privada?

Al declarar la variable se crea una copia en cada hilo, siendo cada hilo el

responsable de uso de esa copia. Al ejecutar **omp2** podemos observar cómo los valores de las variables privadas cambian.

Salida del programa o

```
[arqo102@labomat36 ej1]$ make omp2
gcc -g -lgomp -lm -Wall -D_GNU_SOURCE -fopenmp  -o omp2 omp2.c
[arqo102@labomat36 ej1]$ ./omp2
Inicio: a = 1,    b = 2,  c = 3
        &a = 0x7fff83f7376c,x   &b = 0x7fff83f73768,    &c = 0x7fff83f73764

[Hilo 0]-1: a = 0,        b = 2,  c = 3
[Hilo 2]-1: a = 0,        b = 2,  c = 3
[Hilo 2]          &a = 0x7fe30330ce04,    &b = 0x7fff83f73768,    &c = 0x7fe30330ce08
[Hilo 2]-2: a = 15,       b = 4,  c = 3
[Hilo 0]          &a = 0x7fff83f73724,    &b = 0x7fff83f73768,    &c = 0x7fff83f73728
[Hilo 0]-2: a = 21,       b = 6,  c = 3
[Hilo 1]-1: a = 32739,    b = 2,  c = 3
[Hilo 1]          &a = 0x7fe303d0de04,    &b = 0x7fff83f73768,    &c = 0x7fe303d0de08
[Hilo 1]-2: a = 1071842148,       b = 8,  c = 1071842124
[Hilo 3]-1: a = 0,        b = 2,  c = 3
[Hilo 3]          &a = 0x7fe30290be04,    &b = 0x7fff83f73768,    &c = 0x7fe30290be08
[Hilo 3]-2: a = 33,       b = 10,         c = 3

Fin: a = 1,       b = 10,         c = 3
        &a = 0x7fff83f7376c,    &b = 0x7fff83f73768,    &c = 0x7fff83f73764
```

1.5 ¿Qué ocurre con el valor de una variable privada al comenzar a ejecutarse la región paralela?

Mediante el uso de *private*, se hace una copia de las variables “a” y “tid” en cada hilo; también se hace uso de *firstprivate*(c) y se hace una copia, pero con el valor de “c” en hilo maestro (podemos observar que dicho valor no se modifica)

1.6 ¿Qué ocurre con el valor de una variable privada al finalizar la región paralela?

Al acabar la región paralela, los valores de las diferentes variables vuelven a tener el valor asignado en el hilo maestro, y el valor (los valores modificados en la región paralela se descartan).

1.7 ¿Ocurre lo mismo con las variables públicas?

Al tratarse de variables públicas, se realiza una copia única y todos los hilos la pueden modificar, y a diferencia de las privadas, las modificaciones en las

región paralela son efectivas y acaban modificando el valor de la variable en el hilo maestro.

Ejercicio 2: Paralelizar el producto escalar

2.1 Ejecute la versión serie y entienda cual debe ser el resultado para diferentes tamaños de vector.

Este es el resultado de ejecutar el programa producto escalar en serie tal y como nos lo encontramos:

```
[arqo102@labomat36 ej2]$ ./pescalar_serie
Resultado: 1000000.000000
Tiempo: 0.010167
```

Observamos que el resultado obtenido es igual que el tamaño del vector.

2.2 Ejecute el código paralelizado con el pragma openmp y conteste en la memoria a las siguientes preguntas: ¿Es correcto el resultado? ¿Qué puede estar pasando?

```
[arqo102@labomat36 ej2]$ ./pescalar_par1
Se han lanzado 16 hilos.
Resultado: 86076.000000
Tiempo: 0.031580
```

Observamos que el resultado que nos refleja es distinto al de ejecutar el programa anterior, lo cual es un resultado erróneo. La única diferencia que existe entre los dos programas se encuentra en que **pescalar_par1** usa el **pragma omp parallel for**, lo que causa el error en el resultado.

2.3 Modifique el código anterior y denomine el programa pescalar_par2. Esta versión deber dar el resultado correcto utilizando donde corresponda alguno de los siguientes pragmas:
#pragma omp critical

#pragma omp atomic

¿Puede resolverse con ambas directivas? Indique las modificaciones realizadas en cada caso. ¿Cuál es la opción elegida y por qué?

Sí, se puede resolver con ambas directivas. Para **critical** debemos borrar pragma omp parallel for e insertar pragma omp critical justo en la línea

anterior a la suma. Para **atomic**, el funcionamiento es el mismo, quitando pragma omp parallel for e introduciendo pragma omp atomic.

Los resultados obtenidos han sido más rápidos utilizando critical, por lo que hemos optado por esta opción.

```
[arqo102@labomat36 ej2]$ ./pescalar_par2
Se han lanzado 16 hilos.
Resultado: 1000000.000000
Tiempo: 0.030233
```

2.4 Modifique el código anterior y denomine el programa resultante pescalar_par3. Esta versión deber dar el resultado correcto utilizando donde corresponda el pragma:

#pragma omp parallel for reduction

Comparando con el punto anterior ¿Cuál será la opción elegida y por qué?

```
[arqo102@labomat36 ej2]$ ./pescalar_par3
Se han lanzado 16 hilos.
Resultado: 1000000.000000
Tiempo: 0.024448
```

Este pragma es más eficiente en cuanto a tiempo de ejecución, por lo tanto, esta opción sería la idónea.

2.6 Análisis de tiempos de ejecución. En términos del tamaño de los vectores, ¿compensa siempre lanzar hilos para realizar el trabajo en paralelo, o hay casos en los que no? Si no compensara siempre, ¿en qué casos no compensa y por qué?

No siempre compensa lanzar hilos para los trabajos en paralelo. Por ejemplo, si el programa es muy "sencillo", compensa más lanzar menos hilos o incluso solo 1.

¿Se mejora siempre el rendimiento al aumentar el número de hilos a trabajar? Si no fuera así, ¿a qué debe este efecto?

No siempre se mejora el rendimiento, por ejemplo, si se ponen todos los hilos disponibles de los núcleos a trabajar al mismo tiempo, afecta al rendimiento y empeora los resultados.

Ejercicio 3: Paralelizar la multiplicación de matrices

Tiempos de ejecución (s)

Versión\# hilos	1	2	3	4
Serie	54.572744	54.284786	54.424623	53.983230
Paralela – bucle1	62.993074	47.918503	35.339380	27.799993
Paralela – bucle2	66.565114	43.798115	33.204908	20.940539
Paralela – bucle3	66.072007	41.097511	27.560537	22.011391

Speedup **(tomando como referencia la versión serie)**

Versión\# hilos	1	2	3	4
Serie	1			
Paralela – bucle1	0.8663292729	1.1328564667	1.5400559658	1.9418432947
Paralela – bucle2	0.8198400140	1.2394320166	1.6390535700	2.5779293455
Paralela – bucle3	0.8259586242	1.3208777047	1.9747301367	2.4525133372

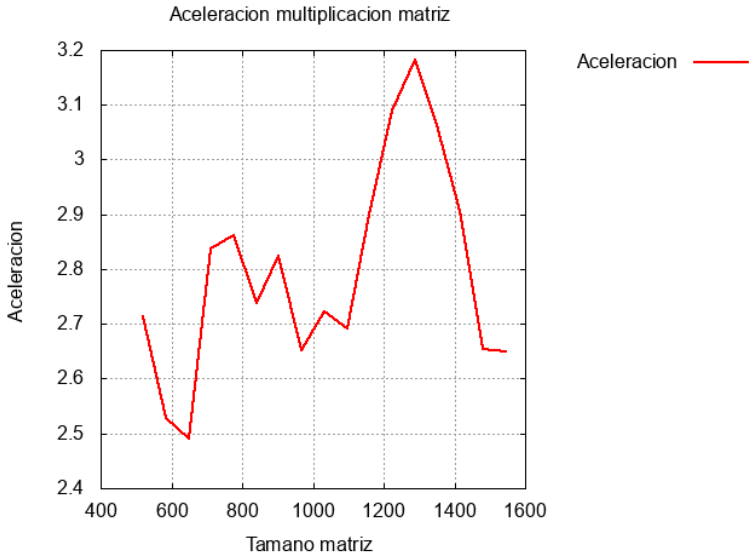
3.1 ¿Cuál de las tres versiones obtiene peor rendimiento? ¿A qué se debe? ¿Cuál de las tres versiones obtiene mejor rendimiento? ¿A qué se debe?

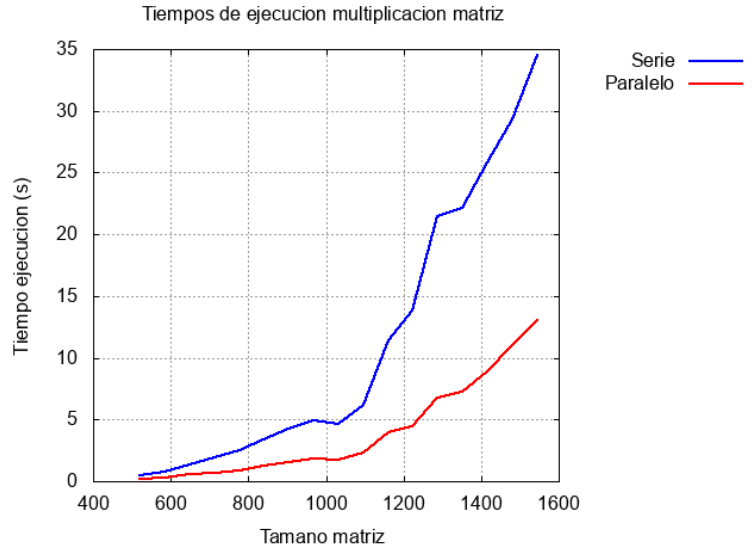
El peor rendimiento lo obtiene la multiplicación de matrices paralela del bucle uno, que, de media, da un tiempo de ejecución en segundos de 43.5127375. Creemos que esto se debe a que, como se paraleliza el bucle más interno del código, tiene que "realizar más esfuerzo" y esto afecta al rendimiento, ya que no aprovecha toda la paralelización al contener bucles externos.

El mejor rendimiento lo obtiene la multiplicación de matrices paralela del bucle tres, que, de media, da un tiempo de ejecución en segundos de 39,1853615. Tal y como hemos comentado en el punto anterior, ahora al estar la paralelización en el bucle más externo, ahora los hilos se lanzan solo una vez (cada uno) y no se destruyen hasta que termina esa ejecución del bucle.

3.2 En base a los resultados, ¿cree que es preferible la paralelización de grano fino (bucle más interno) o de grano grueso (bucle más externo) en otros algoritmos?

Pensamos que a partir del ejemplo que hemos visto en el punto anterior, se aprovecha mas la paralelización de grano grueso que la de grano fino, ya que ahorramos a la máquina el esfuerzo de la creación y destrucción de un mismo hilo todo el tiempo para ejecutar el código.





Comentario sobre las gráficas: En la gráfica del tiempo de ejecución de la matriz, podemos observar que a partir del tamaño 1000 de la matriz, el tiempo de ejecución incrementa considerablemente, en contraparte a la ejecución en paralelo, que incrementa en muchísima menor medida.

Para la aceleración de la matriz observamos una caída en ella cuando llegamos al tamaño de matriz 600, sin embargo, a partir de ese tamaño generalmente la aceleración aumenta, observando un pico al llegar a un tamaño de 1300.

3.3 Si en la gráfica anterior no obtuvo un comportamiento de la aceleración en función de N que se estabilice o decrezca al incrementar el tamaño de la matriz, siga incrementando el valor de N hasta conseguir una gráfica con este comportamiento e indique para que valor de N se empieza a ver el cambio de tendencia.

Como hemos comentado anteriormente y se puede observar en la gráfica, a partir de N=1300 el comportamiento de la aceleración decrece drásticamente.

Ejercicio 4: Ejemplo de integración numérica

Pregunta 4.1: ¿Cuántos rectángulos se utilizan en la versión del programa que se da para realizar la integración numérica?

Para realizar la integración numérica se utilizan n=100000000 rectángulos.

Pregunta 4.2: ¿Qué diferencias observa entre estas dos versiones?

El programa **pi_par1.c** almacena el resultado de las sumas que se realizan con la ejecución del bucle en una variable sum, y el programa **pi_par4.c** utiliza una variable privada del hilo para almacenar la suma de las ejecuciones del bucle.

Ejercicio 4.3: Ejecute las dos versiones recién mencionadas. ¿Se observan diferencias en el resultado obtenido? ¿Y en el rendimiento? Si la respuesta fuera afirmativa, ¿sabría justificar a qué se debe este efecto?

```
[arqo102@labomat36 ej4]$ ./pi_par1
Numero de cores del equipo: 16
Resultado pi: 3.141593
Tiempo 2.093377
[arqo102@labomat36 ej4]$ ./pi_par4
Numero de cores del equipo: 16
Resultado pi: 3.141593
Tiempo 0.307308
[arqo102@labomat36 ej4]$
```

El resultado de ambas ejecuciones es el mismo, pero se observa una clara diferencia en el tiempo necesario para obtener el resultado. Esto se debe a que el programa **pi_par1.c** accede y modifica la variable sum tantas veces como iteraciones tiene el bucle. En cambio, el programa **pi_par4.c** accede solo una vez. Lo ocurrido en el programa **pi_par1.c** se conoce como “False sharing”.

Ejercicio 4.4: Ejecute las versiones paralelas 2 y 3 del programa. ¿Qué ocurre con el resultado y el rendimiento obtenido? ¿Ha ocurrido lo que se esperaba?

```
[arqo102@labomat36 ej4]$ ./pi_par2
Numero de cores del equipo: 16
Resultado pi: 3.141593
Tiempo 2.548003
[arqo102@labomat36 ej4]$ ./pi_par3
Numero de cores del equipo: 16
Double size: 8 bytes
Cache line size: 64 bytes => padding: 8 elementos
Resultado pi: 3.141593
Tiempo 0.305513
[arqo102@labomat36 ej4]$
```

El resultado de ambas ejecuciones vuelve a ser el mismo, y se observa una diferencia en tiempos de ejecución muy parecida a la de los programas anteriores, algo esperable, ya que el código de estos dos nuevos programas es muy parecido a los anteriores. Comparando **pi_par2.c** y **pi_par1.c** , el programa pi_par2.c obtiene un tiempo algo superior que pi_par1.c. Lo que era esperable, ya que con la modificación realizada el problema del “False sharing” persiste. Comparando **pi_par3.c** y **pi_par4.c**, su rendimiento es muy parecido debido a que antes de lanzar los hilos, el hilo maestro se encarga de reservar memoria para el array “sum” de forma que cada índice

del array ocupa un bloque de cache(Cache line size: 64 bytes)

Ejercicio 4.5: Abra el fichero pi_par3.c y modifique la línea 32 del fichero para que tome los valores fijos 1, 2, 4, 6, 7, 8, 9, 10 y 12. Ejecute este programa para cada uno de estos valores. ¿Qué ocurre con el rendimiento que se observa?

```
[arqo102@labomat36 ej4]$ ./pi_par3
Numero de cores del equipo: 16
Double size: 8 bytes
Cache line size: 64 bytes => padding: 1 elementos
Resultado pi: 3.141593
Tiempo 2.082498
[arqo102@labomat36 ej4]$
[arqo102@labomat36 ej4]$ ./pi_par3
Numero de cores del equipo: 16
Double size: 8 bytes
Cache line size: 64 bytes => padding: 2 elementos
Resultado pi: 3.141593
Tiempo 0.950483
[arqo102@labomat36 ej4]$
[arqo102@labomat36 ej4]$ ./pi_par3
Numero de cores del equipo: 16
Double size: 8 bytes
Cache line size: 64 bytes => padding: 4 elementos
Resultado pi: 3.141593
Tiempo 0.521926
[arqo102@labomat36 ej4]$
[arqo102@labomat36 ej4]$ ./pi_par3
Numero de cores del equipo: 16
Double size: 8 bytes
Cache line size: 64 bytes => padding: 6 elementos
Resultado pi: 3.141593
Tiempo 0.460827
[arqo102@labomat36 ej4]$
[arqo102@labomat36 ej4]$ ./pi_par3
Numero de cores del equipo: 16
Double size: 8 bytes
Cache line size: 64 bytes => padding: 7 elementos
Resultado pi: 3.141593
Tiempo 0.503995
[arqo102@labomat36 ej4]$
[arqo102@labomat36 ej4]$ ./pi_par3
Numero de cores del equipo: 16
Double size: 8 bytes
Cache line size: 64 bytes => padding: 8 elementos
Resultado pi: 3.141593
Tiempo 0.180810
[arqo102@labomat36 ej4]$
[arqo102@labomat36 ej4]$ ./pi_par3
Numero de cores del equipo: 16
Double size: 8 bytes
Cache line size: 64 bytes => padding: 9 elementos
Resultado pi: 3.141593
Tiempo 0.181003
[arqo102@labomat36 ej4]$
[arqo102@labomat36 ej4]$ ./pi_par3
Numero de cores del equipo: 16
Double size: 8 bytes
Cache line size: 64 bytes => padding: 10 elementos
Resultado pi: 3.141593
Tiempo 0.306304
[arqo102@labomat36 ej4]$
[arqo102@labomat36 ej4]$ ./pi_par3
Numero de cores del equipo: 16
Double size: 8 bytes
Cache line size: 64 bytes => padding: 12 elementos
Resultado pi: 3.141593
Tiempo 0.307191
[arqo102@labomat36 ej4]$
```

Podemos comprobar que el rendimiento mejora a medida que vamos aumentando el valor de la variable modificada **padsz**. Esto es lógico, ya que al aumentar el padding aumentamos el tamaño de cada posición del array sum, evitando el "False sharing". El mejor rendimiento lo obtenemos cuando el padding es igual a 8 (el mismo padding que se aplicaba sin modificar la variable padsz). Esto se debe a que justamente 8 es el tamaño del bloque.

Ejercicio 4.6: Ejecute las versiones 4 y 5 del programa. Explique el efecto de utilizar la directiva critical. ¿Qué diferencias de rendimiento se aprecian? ¿A qué se debe este efecto?

```
[arqo102@labomat36 ej4]$ ./pi_par4
Numero de cores del equipo: 16
Resultado pi: 3.141593
Tiempo 0.301510
[arqo102@labomat36 ej4]$ ./pi_par5
Resultado pi: 2.113131
Tiempo 4.006851
[arqo102@labomat36 ej4]$
```

Podemos observar que el programa **pi_par5.c** ha aumentado el tiempo de ejecución con respecto a **pi_par4.c**, además de dar un resultado erróneo cada vez que se ejecuta. Con el efecto de la directiva **critical** en el programa pi_par5.c, se restringe la ejecución de la operación "pi += sum;" a un único hilo cada vez. Esto hace que la ejecución del programa se ralentice, ya que pueden producirse esperas para escribir en la variable si varios hilos desean hacerlo al mismo tiempo. Además, y como hemos comentado anteriormente, el resultado del programa pi_par5.c es incorrecto debido a que la variable "i" que se utiliza como contador en el bucle de la región paralela es una variable compartida entre todos los hilos.

Ejercicio 4.7: Ejecute las versiones 6 y 7 del programa. Explique el efecto de utilizar las directivas utilizadas. ¿Qué diferencias de rendimiento se aprecian? ¿A qué se debe este efecto?

```
[arqo102@labomat36 ej4]$ ./pi_par6
Numero de cores del equipo: 16
Resultado pi: 3.141593
Tiempo 2.352316
[arqo102@labomat36 ej4]$ ./pi_par7
Resultado pi: 3.141593
Tiempo 0.299959
[arqo102@labomat36 ej4]$
```

Podemos observar que el programa **pi_par6.c** utiliza un bucle for para automatizar el reparto de la ejecución de las iteraciones de un bucle entre todos los hilos creados de la región paralela. Sin embargo, aparece "False sharing", y este problema hace que el rendimiento se vea afectado con respecto a pi_par7.c. Por otro lado, el programa **pi_par7.c** utiliza la cláusula **reduction**, la cual se utiliza con la variable "sum" y permite que se efectúen sobre ésta operaciones de acumulación en modo atómico. Esto hace que se eviten errores en la operación de suma entre los distintos hilos al tratar de escribir sobre la misma variable al mismo tiempo, y la propia cláusula realiza la "acumulación" sobre la variable "sum" de forma correcta.