MEMORIA PRACTICA 2

CARLOS HERRERO BARBERA | MANEL LURBE SEMPERE

ÍNDICE

En esta memoria, acerca de la práctica 2 de la asignatura Computación Paralela, se van a explicar las conclusiones que hemos sacado al experimentar con la API OpenMP para la programación multi-hilo de memoria compartida, utilizando para ello el clúster de computación "Kahan" del departamento DSIC de la Universidad Politécnica de Valencia.

Las pruebas se han realizado con un programa que calcula la cantidad de divisores exactos que tienen cada uno de los M primeros números naturales, con la intención de mostrar los N números enteros con una mayor cantidad de divisores, en todos los casos se ha instanciado M como 100000 y N como 10.

1. Ejercicio 1

El primer ejercicio consiste en medir el tiempo de ejecución del programa, para ello se ha utilizado la función "omp_get_wtime()" que proporciona OpenMP. Esta función devuelve un valor de tipo "double" con el tiempo del sistema en el momento en que es invocada.

```
...
double tmp = omp_get_wtime();

for ( n = 1 ; n <= M ; n++ ) {
  ...
}

tmp = omp_get_wtime() - tmp;
printf("Time on parallel: %5.5f\n", tmp);
```

Figura 1: Medidas de tiempo de ejecución (divi-e1.c)

Para tomar la medida del tiempo, se debe invocar a esta función justo antes y después del bucle con el que vamos a trabajar (como se indica en la Figura 1), de esta forma ajustamos la medida solo a esta región de código. Además, una vez paralelizado debemos tomar las medidas fuera de la región paralela, ya que en caso contrario el tiempo sería erróneo.

Tiempo Secuencial
41,56011

Figura 2: Tiempo de ejecución en secuencial.

2. Ejercicio 2

Este ejercicio consiste en paralelizar los tres bucles por separado y comparar el speedup y la eficiencia de cada uno de ellos.

a. **Bucle 1** (divi-e2-p1)

Este bucle se ha paralelizado usando la directiva "parallel for" de OpenMP justo antes del bucle "for" que queremos paralelizar (tal como se indica en la Figura 2), esta directiva nos permite la paralelización de un bucle "for" de forma automática, siempre y cuando este bien delimitado, indicando además el tipo de planificación deseado.

En este caso necesitamos indicar que variables serán privadas, esto significa que cada uno de los hilos tendrá su propia copia y de esta forma evitamos las condiciones de carrera, también es necesario usar la directiva "critical" ya que observamos dependencias de datos al acceder concurrentemente a los vectores "vc" y "vn".

```
...
#pragma omp parallel for private(c, i, ini, inc) schedule(runtime)
for ( n = 1 ; n <= M ; n++ ) {
  ...
  if ( MEJOR(n, c, N-1) ) {
    #pragma omp critical
    if ( MEJOR(n, c, N-1) ) {
      for ( i = N - 1; i > 0 && MEJOR(n, c, i-1) ; i-- ) {
        vc[i] = vc[i-1]; vn[i] = vn[i-1];
      }
      vc[i] = c; vn[i] = n;
    }
  }
}
```

Figura 3: Paralelización del bucle 1 (divi-e2-p1.c)

A continuación, se presentan los resultados obtenidos de la ejecución del código mostrado anteriormente, con diferentes planificaciones y distinto número de hilos.

Hilos	Static	Static 1	Dynamic	Guided
2	30,48225	25,77254	19,58993	19,57444
4	17,88133	12,90135	9,84399	9,8159
8	9,58189	6,82279	5,26922	5,25899
16	5,71771	4,04512	3,06395	3,12538
32	3,60604	2,55361	1,91707	1,91553

Figura 4: Tiempos de ejecución.

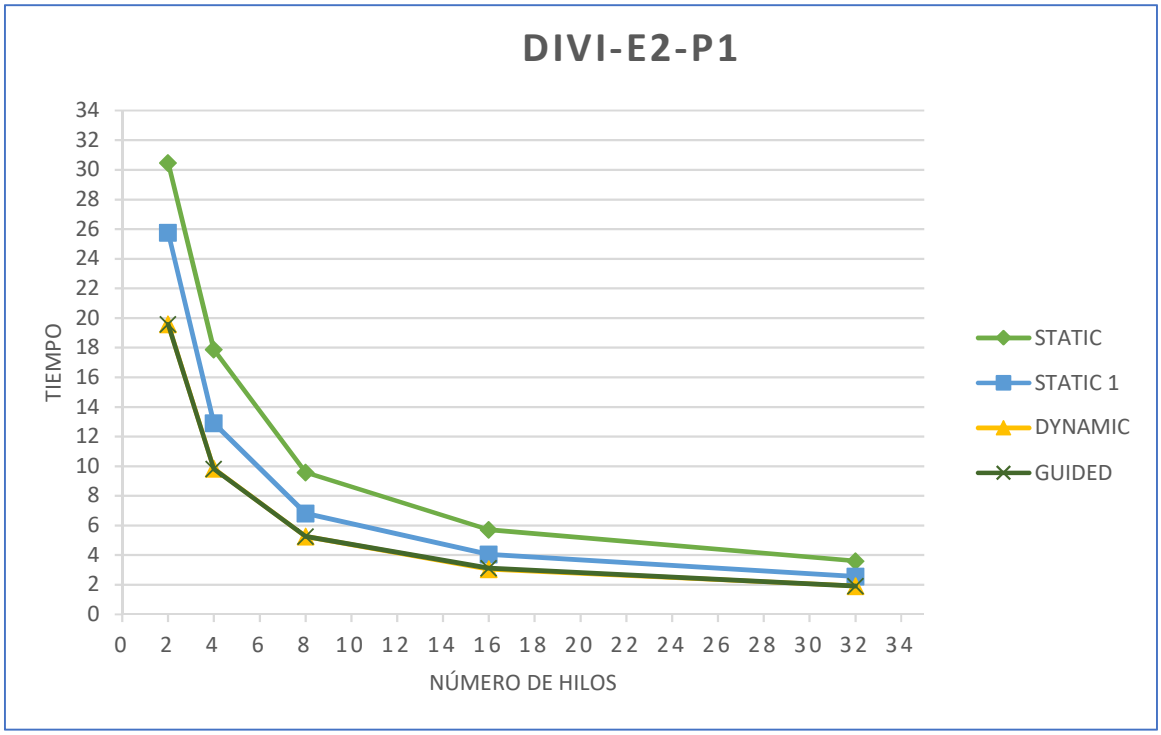

Figura 5: Representación de los tiempos de ejecución del bucle 1.

Aquí podemos observar claramente, que la mejor planificación es la dinámica (dynamic o guided), esto se debe, a que la carga de trabajo es diferente en cada uno de los hilos, y por tanto, el reparto de trabajo en tiempo de ejecución resulta más eficiente.

Respecto al speedup, sí que se aprecia una diferencia significativa en el aumento de hilos, entre 2 y 8 hilos, con relación al tiempo de ejecución, llegados a este punto, la diferencia entre 8 y 32 hilos ya no sería tan significativa.

b. Bucle 2 (divi-e2-p2)

Por lo que respecta a este bucle, la paralelización es mucho más sencilla, debido a que solo tenemos dos variables compartidas que crean conflictos de datos. Por una parte, está la variable "i", la cual, al ser el índice del bucle, la directiva "for" la interpreta como privada implícitamente. Por otro lado, tenemos la variable "c" cuya finalidad es almacenar la cantidad de divisores de cada número.

Para esta tarea se ha utilizado la directiva "reduction" que nos proporciona OpenMP. La directiva "reduction(operación:variable)" especifica que la "variable" es privada, realizando una copia de la misma en cada hilo, y tras finalizar la ejecución de la región paralela, realiza la "operación" sobre la "variable", quedándose con el resultado en el hilo principal del programa.

Para ser más concreto, en este caso la operación a realizar es una suma, por tanto, la directiva "reduction" creará una variable c en cada hilo y al finalizar la ejecución del bucle cada hilo sumará el resultado obtenido con el del hilo principal, quedándoos este último con el resultado.

```
...
#pragma omp parallel for reduction(+:c) schedule(runtime)
for ( i = ini ; i <= n ; i += inc ) {
   if ( n % i == 0 ) c++;
}
...
```

Figura 6: Paralelización del bucle 2 (divi-e2-p2.c)

Pasamos ahora a la presentación de los resultados obtenidos en la ejecución de este código. En la figura 7, que veremos a continuación, hemos decidido no representar el tiempo de la planificación "dynamic" debido a que los tiempos son muy elevados con relación a las otras planificaciones y descuadraba el grafico.

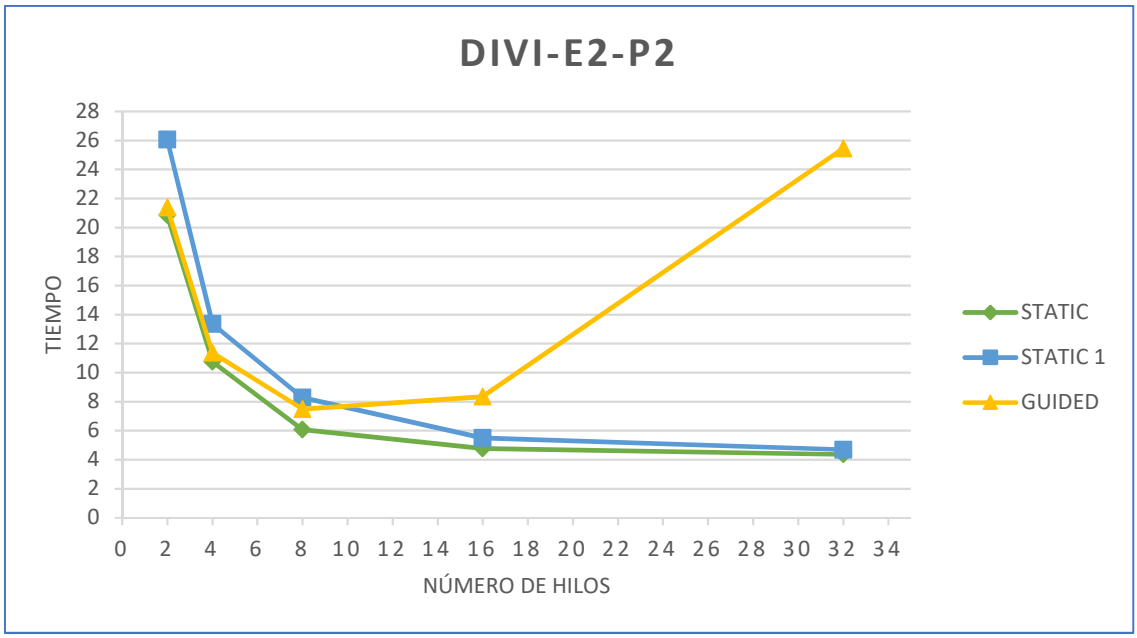

Figura 7: Representación de los tiempos de ejecución del bucle 2.

Hilos	Static	Static 1	Guided	Dynamic
2	20,84654	26,06225	21,37712	322,4874
4	10,75871	13,34366	11,36034	370,90972
8	6,06485	8,28698	7,48703	364,2118
16	4,75958	5,49992	8,34926	477,76997
32	4,36401	4,69187	25,44611	603,31789

Figura 8: Tiempos de ejecución del bucle 2.

En este caso se observa que la planificación estática es mucho mejor. Esto se debe a que el coste de procesar una iteración de este bucle es mínimo y siempre el mismo, por tanto, hacer uso de planificaciones dinámicas requiere un consumo innecesario de recursos para la asignación de las iteraciones en tiempo de ejecución, por esto en la planificación

"dynamic" y en "guided" con 32 hilos se dedica más tiempo a la asignación de las iteraciones a los hilos que al propio problema.

c. **Bucle 3** (divi-e2-p3)

Por lo que respecta a este ejercicio, no se puede paralelizar. Como podemos observar en la Figura 9, la delimitación del bucle incluye una condición obtenida de la invocación a la función "MEJOR", esto hace que la talla del problema tenga instancias significativas y por tanto, que no se pueda hacer un reparto de la carga de trabajo en diferentes hilo, ya que si una iteración anterior no cumpliera la condición del bucle, este hilo detendría su ejecución pero el resto no, y daría un resultado erróneo.

```
...
if ( MEJOR(n,c,N-1) ) {
  for ( i = N - 1 ; i > 0 && MEJOR(n,c,i-1) ; i-- ) {
    vc[i] = vc[i-1]; vn[i] = vn[i-1];
  }
  vc[i] = c; vn[i] = n;
}
```

Figura 9: Paralelización del bucle 3 (divi-e2-p3.c)

d. Conclusiones

Para concluir con este ejercicio, se presentan a continuación varios gráficos que representan el speedup y la eficiencia de cada uno de los apartados del ejercicio. Todos estos datos se han obtenido comparando el tiempo de la planificación más eficiente de cada apartado con el tiempo del algoritmo en secuencial. Para el primer apartado se ha escogido la planificación "dynamic" y en el segundo "static".

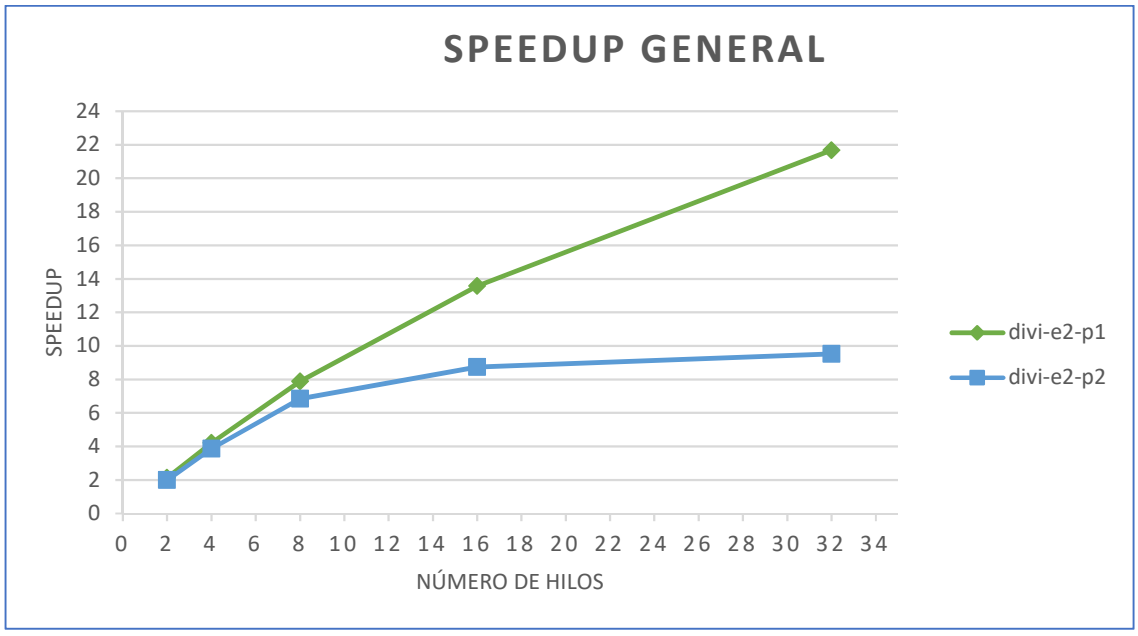

Figura 10: Representación del speedup de los bucles 1 y 2.

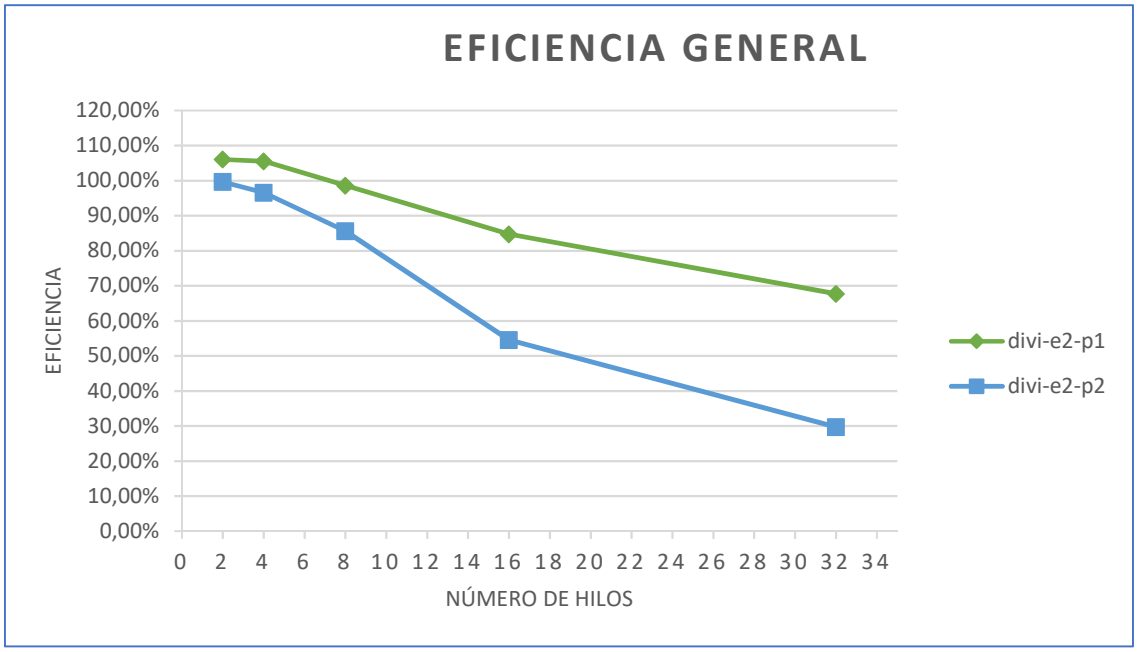

Figura 11: Representación de la eficiencia de los bucles 1 y 2.

Por lo que respecta al speedup, se observa que en los dos bucles tenemos un speedup máximo, para 2 y 4 hilos, sin embargo, a medida que se aumenta el número de hilos, la eficiencia disminuye debido a que no se están aprovechando al máximo cada uno de los hilos, esto se debe a que la paralelización del bucle está llegando a su límite.

3. Ejercicio 3

Tal y como se ha dicho en el ejercicio anterior, el mejor resultado lo hemos obtenido con la paralelización del primer bucle, usando una planificación dinámica, por consiguiente, hemos usado este bucle para la realización del ejercicio 3.

En este ejercicio se pide modificar el programa descrito anteriormente para mostrar por pantalla la cantidad de divisores que ha encontrado cada hilo.

Para realizar esta tarea, hemos decidido crear una región paralela que abarque todo el bucle, inicializando dos variables en su interior, ambas indicadas como privadas para que cada hilo tenga una copia diferente. Una primera variable “cont”, en la cual cada hilo ira sumando los divisores que encuentra, y otra “numfil” donde se almacena el identificador del hilo. Por último, usamos la función “printf” dentro de la región paralela para que cada hilo muestre sus resultados.

Por otra parte, tal y como ya se observó en el ejercicio anterior la paralelizacion se ha realizado con la directiva “for” y además se ha usado “critical” para el control del acceso concurrente a los vectores públicos.

```
...
int cont, numfil;
#pragma omp parallel private(cont, numfil)
{
 cont = 0; numfil = omp_get_thread_num();

 #pragma omp for private(c, i, ini, inc) schedule(runtime)
 for ( n = 1 ; n <= M ; n++ ) {
  ...
  for ( i = ini ; i <= n ; i += inc ) {
   if ( n % i == 0 ) { c++; cont++;}
  }

  if ( MEJOR(n, c, N-1) ) {
   #pragma omp critical
   if ( MEJOR(n, c, N-1) ) {
    ...
  }
 }
```

Figura 12: Contar divisores que encuentra cada hilo (fichero divi-e3.c

4. Ejercicio 4

Este ejercicio consiste en paralelizar con una región paralela el bucle del Ejercicio 2 que mejores resultados nos haya dado, repartiendo nosotros las iteraciones del mismo. Para esto, como ya se ha explicado anteriormente hemos elegido el bucle 1 con una planificación dinámica.

Como se puede observar en la Figura 13, para realizar este ejercicio hemos utilizado tres directivas. La primera "parallel", en la cual indicamos la región de código paralela, por supuesto usando la directiva "private" para especificar que variables serán privadas dentro de esta región.

Por otra parte, también se ha utilizado la directiva "critical", esta se ha usado dos veces, por tanto, debemos indicar el nombre a cada una de ellas. La primera "mas", evita las condiciones de carrera que podrían ocurrir al modificar concurrentemente los vector "vc" y "vn", no vamos a entrar en más detalles ya que se explicó en su momento en el Ejercicio 2. Por lo que respecta al segundo uso de la directiva "critical", evita que dos hilos obtengan una misma iteración del bucle.

Además, hemos utilizado dos variables nuevas. La primera, "fil", privada e inicializada con el identificador de cada hilo obtenido invocado a la función "omp_get_thread_num()" dentro de la región paralela y antes de entrar en el bucle. Esta función forma parte de la API de OpenMP y nos devuelve el identificador del hilo que la invoca. La segunda variable, "pos", será la encargada de llevar la cuenta de las iteraciones que se han realizado hasta el momento y la utilizaremos para indicar la siguiente iteración que tendrá que hacer cada hilo, esta variable la inicializamos invocando a la función "omp_get_num_threads()" que nos devuelve el número de hilos utilizados.

```
...
#pragma omp parallel private(n, c, ini, inc, i, fil)
{
   fil = omp_get_thread_num();
   pos = omp_get_num_threads() + 1;
   n = fil + 1;

   while (n <= M) {
      ...
      if ( MEJOR(n,c,N-1) ) {
         #pragma omp critical (mas)
         if ( MEJOR(n,c,N-1) ) { ... }
      }
      #pragma omp critical (nextit)
      n = pos++;
   }
```

Figura 13: Paralelización del bucle 1 con una región paralela y una planificación dinámica (fichero divi-e4.c)