PROYECTO GLOBAL

Mikel Fajardo

Índice

1.Resumen y Objetivos

El proyecto básicamente consta en paralelizar dos aplicaciones en C. Contamos con el compilador gcc. La aplicación está enfocada a los sistemas multiprocesador (SMP) y utilizaremos la interfaz 'OpenMP' para la paralelización. Para ello crearemos una nueva versión 'paralela' donde aplicaremos la paralelización, con el objetivo de acelerar la versión en serie.

2.Fundamentos Teóricos

Los conocimientos necesarios para tener en cuenta a la hora de empezar en este proyecto relativos a la paralelización:

- Al ejecutar un programa en paralelo, hay fracciones del programa que deben ejecutarse en serie. Si la fracción en serie aumenta de tamaño a la vez que el problema lo hace, el paralelismo que se puede conseguir estará limitado. Por contra, si a la fracción en serie no le afecta el aumento del tamaño del problema, se usará el paralelismo para resolver problemas de gran tamaño.
- Los sistemas multiprocesador, en los procesos paralelos utilizan variables compartidas para compartir la memoria.

Por eso, se utilizan procesos para sincronizar estas variables, tales como; secciones críticas: fragmentos del código paralelo que son ejecutados por todos los procesadores, pero en serie, barreras: los procesos esperan en un punto específico del programa, hasta que todos los procesos hayan llegado al mismo punto.

- A la hora de repartir el trabajo en una sección paralela, el uso de un reparto estático asegura que no se producirán sobrecargas en la ejecución de los procesos, pero no asegura un reparto equilibrado del trabajo.

Por eso, se usa un reparto dinámico y en éste hay dos variables; chunk scheduling: todas las iteraciones se reparten en chunks del mismo tamaño (prioriza el tiempo de compilación), guided-scheduling: los chunks no son todos del mismo tamaño, empiezan siendo chunks con un gran número de iteraciones, y cada vez los chunks van menguando (prioriza tiempo de ejecución).

3. Aplicación

Ejercicio 1: mcol.c

Se trata de procesar columnas de una matriz, para obtener como resultado la suma de las exponenciales de los elementos de la columna que son mayores que 0. Un fichero de entrada, f1, especifica las columnas (peticiones) que hay que procesar. El programa inicia la matriz a unos valores determinados, lee del fichero de entrada la relación de columnas a procesar, llama a la función que procesa las columnas, y calcula el tiempo de ejecución. Finalmente, se escriben los resultados en el fichero f1_sol.

Antes de empezar la paralelización, debemos calcular los tiempos de ejecución de la versión en serie. A la hora de documentar los tiempos tendremos en cuenta; el máximo, el mínimo, la media y la desviación típica. Aunque a la hora de compararlos con los tiempos paralelizados sólo nos fijaremos en la media.

Serie

MAX	MIN	MEDIA	DES.TÍPICA
1924,395ms	1908,91ms	1920,129ms	6,405ms

Esta es la parte del código que se nos ha pedido para paralelizar.

```
void procesar_peticiones (double mat[NFIL][NCOL], int *columnas, int cant, double
*resultados) {
  int  i, j, columna;
  double y, suma;
  for (j=0; j<cant; j++){
    columna = columnas[j];
    suma = 0.0;
    #pragma omp parallel for schedule(static) private (i,y) reduction (+:suma)
    for (i=0; i<NFIL; i++){
      y = mat[i][columna];
      if (y > 0) suma += exp (y / 100);
    }
    resultados[j] = suma;
  }
}
```

Lo primero de todo decidí paralelizar el segundo bucle ya que me percaté de que en el segundo bucle “for” se accedía a la matriz “mat” por filas, teniendo paralelizado el primer “for”, cada hilo se encargaría de varias columnas. En cambio, si paralelizamos el segundo “for” cada hilo se encargará de varias filas. Con esto reducimos el tiempo de ejecución reduciendo los fallos al acceder a la memoria cache.

En cuanto al reparto de las iteraciones, me he decantado por usar un reparto estático, aunque a primera viste pensé en usar un reparto dinámico debido a que hay un “if” y eso puede condicionar a que algunas iteraciones sean menos costosas que otras. Después de hacer algunas pruebas, concluí con que es más eficiente un reparto estático ya que la operación realizada después del “if” no es muy costosa y en tiempo de ejecución no hay ninguna mejora significativa, todo lo contrario, la sobrecarga del reparto dinámico hace que el tiempo suba excesivamente. He usado un reparto entrelazado, pero también se podría usar un reparto

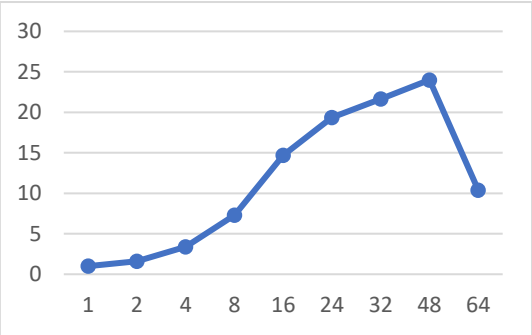
consecutivo, en este caso, no hay ninguna diferencia siempre que el reparto consecutivo se haga bien y se reparta un numero similar de iteraciones a cada hilo.

He puesto las variables “i” e “y” privadas porque necesitamos que cada hilo tenga un valor diferente de estas variables para no alterar los resultados y mejorar el tiempo de ejecución. En el caso de la variable “suma” será “reduction” ya que se esta haciendo una suma a si mismo en cada iteración y necesitamos que se haga esta operación de suma mediante un “reduction”. Si usásemos “private” al final del bucle “suma” no tendría valor y si usamos “shared” se alterará constantemente su valor porque se estarían haciendo varias sumas en paralelo. Y por último la variable “mat” es compartida porque no se hace ningún cambio en la variable y tenemos que sacar los datos que ya están, necesitamos que todos los hilos puedan acceder a los mismos valores de “mat”. Aunque no especifique nada la variable viene predefinida como compartida a no ser que indique lo contrario.

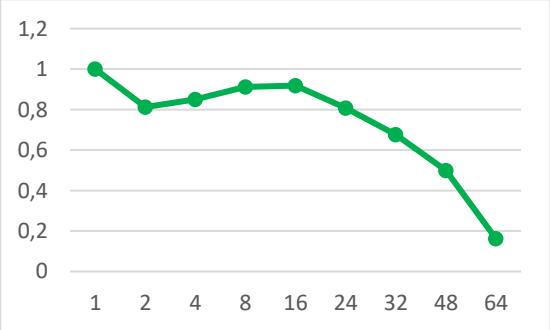
Estos son los resultados obtenidos tras la paralelización:

Hilos	T(ms)	fa	ef
1	1920,129	1,000	1,000
2	1182,919	1,623	0,812
4	564,860	3,399	0,850
8	263,272	7,293	0,912
16	130,759	14,684	0,918
24	99,141	19,368	0,807
32	88,664	21,656	0,677
48	79,990	24,005	0,500
64	185,238	10,366	0,162

1. Gráfico del factor de aceleración de mcol.c



2. Gráfico de la eficiencia de mcol.c



Podemos observar en la tabla que el tiempo de ejecución va bajando a medida que subimos los hilos hasta los 64 hilos donde el tiempo aumenta drásticamente. Esto se debe a que a medida que vamos aumentando los hilos, la sobrecarga sube, ya que la sobrecarga es función de los hilos que se usan, por lo cual, cuando llegamos a los 64 hilos, la sobrecarga es tan grande que hace que el tiempo de ejecución suba, limitando los hilos que podemos usar en 48.

En la gráfica del factor de aceleración podemos ver como sube en función de los hilos hasta llegar a los 64 hilos donde se produce esa sobrecarga excesiva donde baja el factor de aceleración drásticamente. La curva va subiendo cada vez mas hasta los 16 hilos donde se ve como cada vez sube menos hasta los 48 hilos. Esto se debe a que la sobrecarga aumenta

cuando aumentamos los hilos, el cual hace que cada vez aumente menos el factor de aceleración hasta llegar a un punto donde baje, en este caso cuando llega a 64 hilos.

Esto se ve claramente en la gráfica de la eficiencia, donde podemos ver como la eficiencia sube hasta llegar a su punto más álgido en los 16 hilos y a partir de ahí empieza a bajar cada vez más. Con este comportamiento podemos decir que tiene una escalabilidad débil, ya que al aumentar los hilos sin aumentar el tamaño del problema (N) la eficiencia no se mantiene. Con esto podemos deducir que si aumentásemos el tamaño del problema la eficiencia se mantendría si aumentamos a la vez los hilos.

Estos son los mejores tiempos ejecutando el código en paralelo:

Paralelo (48 hilos)

MAX	MIN	MEDIA	DES.TÍPICA
80,99ms	76,934ms	79,99ms	1,74ms

Ejercicio 2: histo.c

En el programa histo.c se procesa una matriz para realizar las siguientes operaciones:

a) Calcular el histograma de la imagen, b) Calcular el valor mínimo del histograma y su posición, c) Calcular la suma de cada fila de la matriz, d) Calcular la suma de cada columna de la matriz.

Antes de empezar la paralelización, debemos calcular los tiempos de ejecución de la versión en serie. A la hora de documentar los tiempos tendremos en cuenta; el máximo, el mínimo, la media y la desviación típica. Aunque a la hora de compararlos con los tiempos paralelizados sólo nos fijaremos en la media.

Estos son los tiempos ejecutando el código en serie:

Serie

MAX	MIN	MEDIA	DES.TÍPICA
248,447ms	247,792ms	247,996ms	0,26ms

Esta es la parte del código que se nos ha pedido para paralelizar.

```
#pragma omp parallel private(i,j,nth) firstprivate(histo_p) shared(histo,imin,hmin,mat)
  {
    nth = omp_get_num_threads ();
    #pragma omp for schedule(static)
    for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      histo_p[mat[i][j]]++;      // a) histograma
    #pragma omp for schedule(static) nowait
    for(j=0; j<nth; j++)
    for(i=0; i<MAX; i++)
      #pragma omp atomic
      histo[i] += histo_p[i];
```

```
#pragma omp single
{hmin = N*N + 1;}
#pragma omp for schedule(static) reduction(min:hmin) nowait
for (i=0; i<MAX; i++){    // b) el valor mínimo de la histograma y su posicion
  if (hmin > histo[i]){
  hmin = histo[i];
  imin = i;
  }
}
#pragma omp for schedule(static) reduction(+:suma_fil) nowait
for (i=0; i<N; i++)    // c) suma de las filas de la matriz
  for (j=0; j<N; j++)
    suma_fil[i] += mat[i][j];
#pragma omp for schedule(static) reduction(+:suma_col)
for (j=0; j<N; j++) // d) suma de las columnas de la matriz
  for (i=0; i<N; i++)
    suma_col[j] += mat[i][j];

}
```

El ejercicio nos pedía que utilizásemos solo una región paralela, al abrir la región pongo las variables “i”, “j” y “nth” como privadas ya que no necesitamos que un hilo sepa los cambios que se producen a esas variables en otro hilo. Pongo la variable “histo_p”, que es una variable auxiliar que he creado para poder calcular el histograma sin utilizar “reduction”, como “firstprivate” para que la variable sea privada y no nos altere los valores al hacerse operaciones en paralelo sobre esa variable y para que todos los hilos accedan al primer valor de esa variable. Las demás variables son compartidas dado que necesitamos que todos los hilos puedan acceder a los valores de esas variables.

A) Para paralelizar el cálculo del histograma se nos pedía que no usásemos variables de tipo “reduction”, entonces para calcularlo he usado el siguiente método. Primero calculamos el histograma mediante dos “for” paralelizando el primero, ya que el acceso a la matriz es por columnas, y metemos los resultados en la variable “histo_p”, que es de tipo “firstprivate” por lo cual cada hilo hará sus operaciones partiendo desde el primer valor de la variable. Como la variable esta definida fuera del “for” cada hilo cojera su primer valor, pero se quedará con un valor, ya que seguiremos dentro de la región paralela, que será el valor que debería de tener si hiciésemos el bucle las iteraciones entre los hilos. Después de eso hice otros dos “for”, el primero para que haga las mismas iteraciones que los hilos estamos usando, y el segundo, para sumar los valores de “histo_p” a los valores de “histo”. Para hacer esa operación usaremos “atomic” para que todos los hilos hagan esa operación y evitar alteraciones en el resultado. Paralelice el primer “for” para que cada hilo se encargase de una iteración del primer “for” y le puse un nowait para romper la barrera que hay al final para que el primer hilo en terminar siguiese haciendo lo demás ya que este bucle es independiente a los que siguen. Hacer esto equivale a paralelizar el bucle usando la variable de tipo “reduction”. He utilizado un “nowait” para romper la barrera ya que el siguiente bucle es independiente de este.
B) Para paralelizar el cálculo del valor mínimo del histograma y su posición, primera le daremos a la variable “hmin”, que será donde guardaremos el valor mínimo, el valor

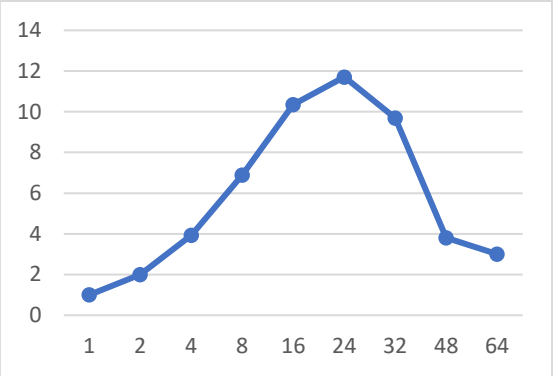
mas alto posible mas uno. Para hacer esta operación, al ser una operación sencilla y poco costosa, lo haremos en serie usando “single” y así también evitaremos mediante las barreras que cualquier hilo avance, dado que la siguiente operación depende del valor de esta variable. El siguiente bucle es el que se encarga de calcular valor mínimo del histograma y su posición, lo paralelice con un reparto estático, aunque a primera vista podríamos hacerlo dinámicamente ya que al haber un “if” no todas las iteraciones tendrán el mismo tiempo de ejecución, al ser una operación tan sencilla la diferencia no es significativa. Además de eso al ser un bucle de solo diez iteraciones, al superar los diez hilos, cada hilo haría una iteración por lo cual daría igual el reparto que usáramos. Use la variable de tipo “reduction” en “hmin” para no alterar el resultado al estar trabajando con una variable compartida en paralelo y así asegurar que se guardase el valor mínimo en esta variable.

C) Para paralelizar el cálculo de la suma de cada fila de la matriz, he paralelizado el primer “for”, ya que el acceso a la matriz es por columnas, utilizando un reparto estático, ya que todas las iteraciones tienen un tiempo de ejecución similar y he usado la variable de tipo “reduction” en “suma_fil” para que no haya ninguna alteración en los resultados de esta variable, debido a que se hace una suma sobre si mismo en cada iteración. Uso “nowait” puesto a que los datos que se utilizan en el siguiente bucle no dependen de los que se utilizan aquí.

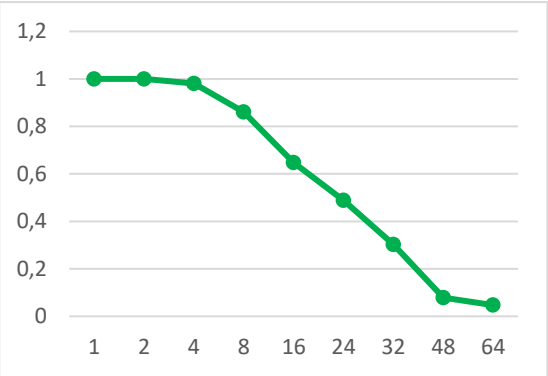
D) Para paralelizar el cálculo de la suma de cada columna de la matriz, he paralelizado el primer “for”, ya que el acceso a la matriz es por columnas, utilizando un reparto estático, ya que todas las iteraciones tienen un tiempo de ejecución similar y he usado la variable de tipo “reduction” en “suma_col” para que no haya ninguna alteración en los resultados de esta variable, debido a que se hace una suma sobre sí mismo en cada iteración.

Hilo	T(ms)	fa	ef
1	247,996	1,000	1,000
2	120,868	2,052	1,026
4	63,174	3,926	0,981
8	35,996	6,890	0,861
16	23,967	10,347	0,647
24	21,180	11,709	0,488
32	25,605	9,685	0,303
48	65,192	3,804	0,079
64	82,551	3,004	0,047

2 Gráfico del factor de aceleración de histo.c



1 Gráfico de la eficiencia de histo.c



Podemos observar en la tabla que el tiempo de ejecución va bajando a medida que subimos los hilos hasta los 32 hilos donde el tiempo aumenta un poco y al llegar a los 48 hilos es donde más aumenta, a partir de ahí el aumento decrece, pero sigue aumentando el tiempo. Esto se debe a que a medida que vamos aumentando los hilos, la sobrecarga sube, ya que la sobrecarga es función de los hilos que se usan, por lo cual, cuando llegamos a los 32 hilos, la sobrecarga es tan grande que hace que el tiempo de ejecución suba, limitando los hilos que podemos usar en 24.

En la gráfica del factor de aceleración podemos ver como sube en función de los hilos hasta llegar a los 32 hilos donde la sobrecarga ya es notoria pero donde más se ve es al llegar a los 48 hilos produciéndose esa sobrecarga excesiva donde baja el factor de aceleración drásticamente. La curva va subiendo cada vez más hasta los 16 hilos donde se ve sube menos hasta los 24 hilos. Esto se debe a que la sobrecarga aumenta cuando aumentamos los hilos, el cual hace que cada vez aumente menos el factor de aceleración hasta llegar a un punto donde baje, en este caso cuando llega a 32 hilos.

Esto se ve claramente en la gráfica de la eficiencia, donde podemos ver como la eficiencia con 2 hilos es ideal, pero de ahí empieza a bajar debido a la sobrecarga generada por los hilos, ejecución que se mantiene en serie y los tiempos de espera generados. Con este comportamiento podemos decir que tiene una escalabilidad débil, ya que al aumentar los hilos sin aumentar el tamaño del problema (N) la eficiencia no se mantiene. Con esto podemos deducir que si aumentásemos el tamaño del problema la eficiencia se mantendría si aumentamos a la vez los hilos.

Estos son los mejores tiempos ejecutando el código en paralelo:

Paralelo (24 hilos)

MAX	MIN	MEDIA	DES.TÍPICA
22,432ms	20,11ms	21,18ms	0,931ms

4. Conclusiones

En cuanto a los objetivos, he logrado reducir considerablemente los tiempos de ejecución manteniendo la funcionabilidad de los programas. He reducido los tiempos todo lo que he podido, por lo cual, estoy satisfecho por el trabajo realizado. Hacer este proyecto me ha ayudado a entender mejor el funcionamiento de la paralelización y saber cual es el funcionamiento de algunos de los códigos usados. En conclusión, he hecho el código y los he explicado lo mejor que he podido, poniendo en este informe los datos con mayor precisión posible y el hacerlo me ha ayudado a entender mejor la paralelización.