Programación Paralela

Rubén García Macho

11 de Diciembre de 2022

Índice

1. Práctica 1

1.1. Ejercicio 1

1.1.1. Implementación con actualizado de variable en el bucle

```
void suma(int* array, int indiceHilo,int N){

    int accesosHilo = trunc(N/Nhilos);//numero de elementos a los que accede el
    hilo
    int i=indiceHilo*accesosHilo;
    int aux = 0;
    while(i<(indiceHilo*accesosHilo+accesosHilo)){

        aux += array[i];
        i++;
        {
            std::lock_guard<std::mutex>guard(g_m);
            sumaTotal+=aux;
        }
    }

    if (indiceHilo == Nhilos && N%Nhilos != 0){
        int i=accesosHilo*(indiceHilo + 1);
        while(i<N){
            aux += array[i];
            i++;
            {
                std::lock_guard<std::mutex>guard(g_m);
                sumaTotal+=aux;
            }
        }

    }

    return;
}
```

Hemos dividido el vector en bloques a los que accederá cada hilo, cuando recorre el vector actualizará la variable en el bucle como se especifica en el enunciado

1.1.2. Tiempo de ejecución entre 1y 16 *Threads*

Hemos utilizado un tamaño de vector de 10000000 para poder observar unas diferencias relevantes entre el número de hilos

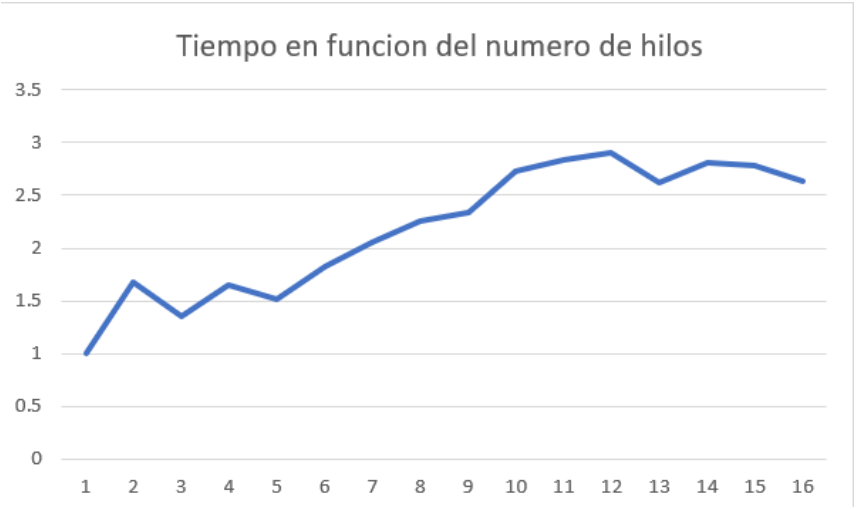

Figura 1: Tiempos de ejecución en función del numero de hilos

1.1.3. Implementación con la variable actualizada 1 vez

```
1
2  void suma(int* array, int indiceHilo,int N){
3
4      int accesosHilo = trunc(N/Nhilos);//numero de elementos a los que accede el
       hilo
5      int i=indiceHilo*accesosHilo;
6      int aux = 0;
7      while(i<(indiceHilo*accesosHilo+accesosHilo)){
8          aux += array[i];
9          i++;
10     }
11
12     if (indiceHilo == Nhilos && N%Nhilos != 0){
13         int i=accesosHilo*(indiceHilo + 1);
14         while(i<N){
15             aux += array[i];
16             i++;
17         }
18
19     }
20
21     {
22         std::lock_guard<std::mutex>guard(g_m);
23         sumaTotal+=aux;
24     }
25
26     return;
27 }
```

Aquí podemos ver que los valores se almacenan en una variable auxiliar con la que más tarde actualizaremos `sumaTotal`. Esto será más óptimo porque solo se bloquea el mutex 1 vez por hilo.

1.1.4. Tiempos en la nueva implementación y observaciones

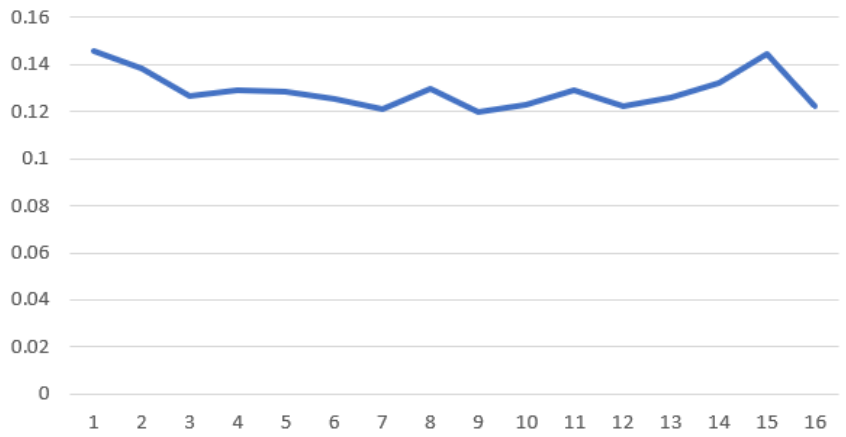

Figura 2: Tiempos de ejecución en función del numero de hilos

Aquí observamos que no hay tanta pérdida de tiempo por lo que al aumentar el número de hilos no se empeora el rendimiento porque no estan ocupandose mutuamente al usar el mutex

1.2. Ejercicio 2

1.2.1. Implementación paralela

```
    for(auto i = 0; i<Nhilos; i++){

        threads[i] = std::thread(rap, array, arrayObj, pedazos[i], Nhilos, i);
    }
    rap(array, arrayObj, N, Nhilos, Nhilos);

    for(auto j = 0; j<Nhilos;j++){
        threads[j].join();
    }
```

Hemos aplicado la fórmula especificada en el anunciado para paralelizarlo, la implementación paralela se ha hecho utilizando hilos síncronos y llamando desde el main a la funcion `rap`.

1.2.2. Gráfica de tiempos de la implementación

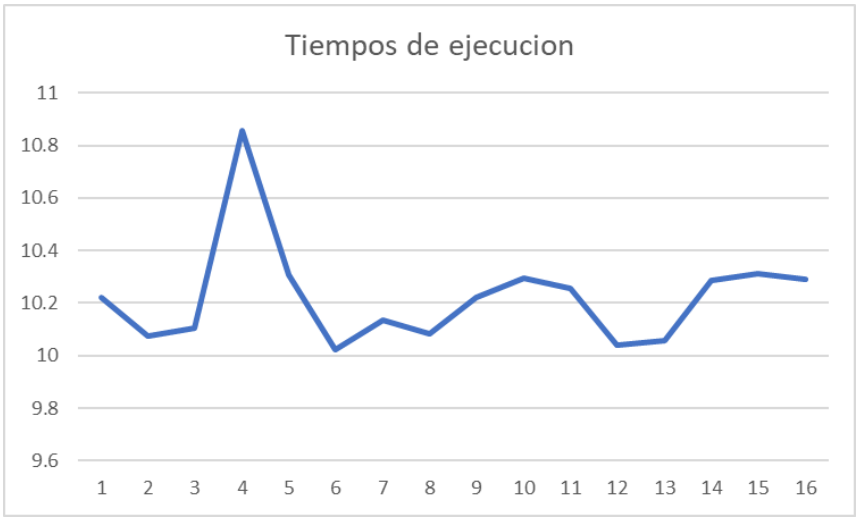

Figura 3: Tiempos de ejecución en función del numero de hilos

1.2.3. Observaciones

En esta gráfica observamos que el tiempo de ejecución aumenta a medida que se utilizan más hilos, esto es porque el problema no se enfoca de una buena forma, no necesariamente este problema sería mas óptimo paralelizándolo, hay otras formas de optimizarlo, como la programación dinámica.

1.3. Ejercicio 3

1.3.1. Trabajo que habría hecho OpenMP y sus ventajas

El trabajo hecho con OpenMP sería como se ha hecho ahora, dividiendo el flujo de trabajo en los distintos hilos incluyendo el main, como se ha hecho en la práctica, también, en OpenMP se declara lo que se quiere paralelizar y OpenMP lo gestiona.

1.3.2. Cómo se haría la implementación a `tasks` de OpenMP

Seria utilizando hilos asíncronos(`detached`)

2. Práctica 2

En esta práctica paralelizaremos el problema de la multiplicación de matrices para que sea mas óptimo que en secuencial. Utilizaremos el código `MatMul_fichero.c` para poder paralelizar el código de forma que podamos tener en el *input* unas matrices de un tamaño lo suficientemente grandes como para poder ver diferencias en los tipos de `schedule` que se observarán en los ejercicios 2 y 3, junto a esta decisión, se ha realizado un código hecho para crear un fichero que tenga 2 matrices cuadradas para que se multipliquen.

2.1. Ejercicio 1

2.1.1. Directivas utilizadas

- `Parallel`: Directiva que se utiliza para declarar el inicio de una región que va a ser paralelizada.
- `for`: Directiva que se coloca delante de un bucle `for` para comenzar a paralelizarlo.
- `omp_get_max_threads`: Obtiene el número máximo de hilos que se pueden utilizar para paralelizarlo bien.
- `omp_get_wtime`: Obtiene el tiempo actual del sistema para hallar los tiempos de ejecución.
- `shared`: función de OpenMP para etiquetar variables como compartidas.
- `private`: función de OpenMP para etiquetar variables como privadas.
- `shchedule`: función que especifica un tipo de reparto de iteraciones en un `for`

2.1.2. Etiquetado de variables

En las funciones hemos etiquetado de forma explícita (`A, B, C, i, j, k`) las demás variables(las correspondientes a las dimensiones) no han sido etiquetadas de forma explícita. Se etiquetarán de forma automática como `shared`

- **Variables compartidas:**Como hemos comentado anteriormente, las variables relacionadas con las dimensiones están etiquetadas de forma automática en `shared` pero las variables A, B y C están etiquetadas de forma manual, esto se hizo por comodidad a la hora de replicar el código a sus distintas funciones. Se han etiquetado de forma compartida porque son variables que se pueden compartir entre sí y no generan ningún problema.

- **Variables privadas:**Hemos etiquetado como variables privadas a las variables que usan los iteradores de los bucles `for` porque por su función necesitan ser privadas.

2.1.3. Reparto de trabajo

En este ejercicio el reparto cumple el paralelismo de datos, compartiendo las iteraciones de un bucle entre los threads del equipo de lo más equitativa posible, en este caso dejándolo en manos del `runtime`

2.1.4. Speedup

Hemos obtenido ejecutando la multiplicación de matrices 17.5891 segundos en paralelo y 68.8948, aplicando la fórmula del speedup:

$$S = \frac{T_1}{T_2}$$
$$S = \frac{68{,}8948}{17{,}5891} = 3{,}9169$$

Esto significa que el programa paralelo se ejecuta 3.9169 veces más rápido, esto es normal porque según la ley de Amdahl no vamos a poder obtener un Speedup igual al numero de hilos con los que se va a paralelizar porque va a haber partes que no se van a poder paralelizar.

2.2. Ejercicio 2

2.2.1. diferencias en `schedule` y los porqués

Hemos observado que hay diferencias en función de que `schedule` hay, el `dynamic` es el más rápido, seguido del `static` y por último el `guided`, esto se produce porque el algoritmo `dynamic` le da "pequeños"trozos a cada hilo, haciendo que se aprovechen al máximo, el `static` tiene alguna pérdida porque asigna bloques de iteraciones y no son del todo iguales y el `guided` es el que peor funciona porque la mayoría de las iteraciones son parecidas.
Hemos tenido un tiempo paralelo sin schedule de 24.8382 s y secuencial de 92.1794 s, por lo que obtenemos un speedup de $S = \frac{92,1794}{24,8382} = 3,711$

2.2.2. Tamaño óptimo en el algoritmo `static`

El tamaño más óptimo para un algoritmo `static` será el mayor tamaño por hilo posible, por ejemplo, si hay un total de 10000 iteraciones entre todos los hilos, lo mejor sería dividir el trabajo en los bloques mas grandes posibles para que haya los menores cambios de contexto.

2.2.3. Tamaño óptimo en algoritmo `dynamic`

Hemos utilizado un script probando tamaños múltiplo de 2 y hemos encontrado que se genera un mayor rendimiento con tamaños de bloque "pequeños"hasta el tamaño de bloque de 32 no se aprecia una gran pérdida de rendimiento.

2.2.4. porqué el algoritmo `guided` tiene los peores tiempos

El algoritmo guided tiene peores tiempos de ejecución porque va variando de tipo de algoritmo y no se reparte el trabajo de una forma correcta.

2.3. Ejercicio 3

2.3.1. Qué algoritmo es el mejor

De forma experimental, hemos obtenido que el mejor algoritmo es el dinámico porque al dividir las iteraciones en múltiples bloques "pequeños.ºbtiene un menor tiempo de ejecución, esto es así porque como el trabajo es parecido entre iteraciones haciendo que el programa acabe cuanto antes.

2.3.2. El peor algoritmo y como mejorar los resultados

El estático es el peor porque por su funcionamiento, a cada hilo se le asigna una cantidad de iteraciones por hilo haciendo que, en las matrices inferiores no se pueda favorecer los tipos de localidades por las tareas asignadas.
Mejoraríamos los resultados dividiendo la matriz en bloques para aprovechar este tipo de schedule.

3. Práctica 3

3.1. Ejercicio 1

3.1.1. Directivas utilizadas

Las nuevas directivas utilizadas son las directivas `single`, `task` y `taskwait`.
La directiva *single* indica un fragmento de código que la va a ejecutar un único hilo(en este caso es la lectura la que se hace de forma no paralela). La directiva *task* indica los fragmentos de código que se van a paralelizar de forma asíncrona. De esta manera, utilizaremos un *taskwait*, que es una barrera, para que no haya problemas en las escrituras.

3.1.2. Etiquetado de variables

Hemos etiquetado como públicas los arrays en los que se encuentran los frames de los vídeos tanto como sus archivos donde se vayan a depositar(in y out) también hemos puesto como públicas las variables asignadas al tamaño de las matrices(height, width y size). Hemos declarado como privada `i` porque es una variable que usaremos en el fragmento de filtrado de vídeo en la que necesitará ser privada.

3.1.3. Como hemos obtenido el paralelismo

```
1   i = 0;
2
3
4      #pragma omp single
5      {
6         do
7         {
8            size = fread(pixels[i], (height+2) * (width+2) * sizeof(int), 1, in);
9
10           if (size)
11           {
12              #pragma omp task
13                 {
14                    fgauss (pixels[i], filtered[i], height, width);
15                 }
16              i++;
17           }
18           if(i >=seq || size == 0){
19              #pragma omp taskwait
20              for (int j = 0;j<i;j++){
21                 fwrite(filtered[j], (height+2) * (width + 2) * sizeof(int), 1,
    out);
22              }
23              i = 0;
24           }
25        } while (!feof(in));
26     }
```

Como vemos en el código, tenemos una región declarada que ejecuta un único hilo, en esta región leemos un fragmento del fichero `in`, después, si el fichero se ha leído bien, tendremos una región paralela asíncrona, esto significa que habrá `seq` hilos accediendo de forma asíncrona a las matrices pixels y filtered. Cada hilo irá modificando `i` por lo que cuando llegue a `i=seq` significa que todos los hilos habrán leído el fichero y se podrá escribir en el nuevo fichero.

3.1.4. Speedup

Hemos obtenido un tiempo de ejecución en secuencial de 3.9000 s y en paralelo de 0.8707 s, siguiendo la formula del *Speedup*

$$S = \frac{3,9}{0,8707} = 4,479$$

4. Práctica 4

4.1. Ejercicio 1

4.1.1. Directivas y funciones de OpenMP

La única directiva nueva que hemos utilizado es `critical`, que crea una sección crítica en su ámbito.

4.1.2. Etiquetado de variables

Hemos etiquetado como variables privadas las variables `i,j,x,y,c` y `n` el código en el que se usan variables privadas hemos etiquetado de forma privada la variable `CPrivada` y las demás variables `(n,count,count_max,x_max,x_min,y_max,y_min,c_max,r,g,b,mut,aux)` Las variables compartidas están siendo compartidas porque necesitan ser comunicadas entre todos los núcleos mientras que las variables privadas no, son principalmente iteradores y variables auxiliares para el desarrollo del código `(x, y, c)`

4.1.3. Paralelismo en este programa

Hemos paralelizado este programa, como se indica el enunciado, paralelizando todo el `main`. Hay secciones que no se pueden paralelizar(porque no necesitan ser paralelizadas, son impresiones por pantalla o interacciones con la memoria, como `mallocs` o escritura en ficheros ppma. Los demás fragmentos son bucles for que hemos paralelizado con la directiva for. La parte que se explica con el comentario *Determine the coloring of each pixel* la explicaremos en el siguiente ejercicio, pues lo requiere el enunciado, esta parte tiene una sección crítica que hay que tratar de una forma específica(*mutexes...*)

4.2. Ejercicio 2: Distintas implementaciones

4.2.1. OpenMP

```
#pragma omp for
  for ( j = 0; j < n; j++ )
  {
  int numeroThread = omp_get_thread_num();
    for ( i = 0; i < n; i++ )
    {
        #pragma omp critical
        {
          if ( c_max < count[i+j*n] )
          {
            c_max = count[i+j*n];
          }
        }

    }
```

En esta variante del código hemos hecho una sección critica con OpenMP, el tiempo de una ejecución de la sección paralela es de 28.79s

4.2.2. Funciones de runtime

```
#pragma omp single
{
  aux=omp_get_num_threads();
  c_max = 0;

    omp_init_lock(&mut);

}

#pragma omp for
  for ( j = 0; j < n; j++ )
  {
  int numeroThread = omp_get_thread_num();
    for ( i = 0; i < n; i++ )
    {

        {
          omp_set_lock(&mut);
          if ( c_max < count[i+j*n]) {
            c_max = count[i+j*n];
          }
          omp_unset_lock(&mut);
        }

    }

  }
/*
  Set the image data.
*/
  #pragma omp single
  {

    omp_destroy_lock(&mut);

    r = ( int * ) malloc ( n * n * sizeof ( int ) );
    g = ( int * ) malloc ( n * n * sizeof ( int ) );
    b = ( int * ) malloc ( n * n * sizeof ( int ) );
```

Aquí hemos inicializado un mutex para realizar la sección crítica, el tiempo de una ejecución de la sección paralela es de 31.28s

4.2.3. Secuencial

```
#pragma omp for
  for ( j = 0; j < n; j++ )
  {
  int numeroThread = omp_get_thread_num();
    for ( i = 0; i < n; i++ )
    {
        if ( c_max < count[i+j*n]) {
          c_max = count[i+j*n];
        }

    }
```

Aquí lo hemos implementado de forma secuencial, el tiempo de una ejecución de la sección sin los fragmento single es de 19.61s.

4.2.4. Variables privadas

```
#pragma omp for
  for ( j = 0; j < n; j++ )
  {
  int numeroThread = omp_get_thread_num();
    for ( i = 0; i < n; i++ )
    {


  if ( CPrivada < count[i+j*n]) {
    CPrivada = count[i+j*n];
  }

  }

  #pragma omp critical
  {
    if (c_max < CPrivada) {
```

Aquí lo hemos implementado mediante una variable privada(CPrivada) que se incrementa en cada hilo, que después mediante un critical ajusta `c_max`, el tiempo de una ejecución de la sección paralela es de 19.91s

4.2.5. Speedups y observaciones

Vamos a realizar los Speedups de los códigos respecto a la implementación secuencial(implementación 3) para evaluar el rendimiento y hallar los porqués de los Speedups.
Hemos medido los tiempos desde la seccion *Carry out the iterartion for each pixel, determining COUT* hasta *Set the image data* incluyéndola.

- $T_{OMP} = 28{,}79s$
- $T_{runtime} = 31{,}28s$
- $T_{seq} = 19{,}61s$
- $T_{Vprivs} = 19{,}91s$

Ahora hallaremos los Speedups:

- $S_{OMP} = 0{,}6811$
- $s_{runtime} = 0{,}6269$
- $S_{seq} = 1$
- $S_{Vprivs} = 0{,}9849$

Hemos hallado que la implementación más óptima es la secuencial, por lo que trabajaremos con ella en os próximos ejercicios.

4.3. Ejercicio 3: Escalado fuerte

4.3.1. Speedup teórico mediante Amdahl

La fórmula del *Speedup* a partir de la ley de Amdahl que usaremos en este apartado es la siguiente:

$$S = \frac{1}{s + \frac{(1-s)}{P}}$$

siendo s la fracción serie, P el numero de procesadores y S el speedup.
Analizando el código hemos hecho el supuesto de que cada linea de código recorrida es equivalente a una operación por comodidad. Hemos hallado la sección serie con los tiempos de la sección paralela y el código total, restando los 2 tiempos tendremos la sección en serie y podremos hallar la fracción serie.
En una ejecución, hemos hallado un tiempo en la región paralela de 19.6124 s y el tiempo total de 31.6662 segundos, de esta forma sabremos que

$$s = \frac{31{,}6662 - 19{,}6124}{31{,}6662} = 0{,}3806$$

Utilizando la fórmula anterior del *speedup* y suponiendo que nuestro computador tiene 8 núcleos, obtendremos un *speedup* de 2.1830 teórico con 8 núcleos, de la siguiente forma hallaremos todos.

- $S_1 = 1$
- $S_2 = 1{,}4485$
- $S_3 = 1{,}7032$
- $S_4 = 1{,}8674$
- $S_5 = 1{,}9820$
- $S_6 = 2{,}0666$
- $S_7 = 2{,}1316$
- $S_8 = 2{,}1830$

4.3.2. ¿Se cumplen los valores teóricos y prácticos?

A continuación adjuntaremos una tabla con los valores teóricos, los prácticos y una gráfica comparándolos.

		Tpractico	Tsecuencial	Spractico	Steorico	Cores
		66.395453	66.057738	0.99491358	1	1
		40.30203		1.63906726	1.44859103	2
31.6662	Total	59.872116		1.1033139	1.70328362	3
19.6124	Paralelo	39.755857		1.66158506	1.86745219	4
12.0538	Serie	47.567809		1.38870676	1.98207593	5
0.38065193	s	38.554146		1.71337573	2.06664259	6
		41.110812		1.60682153	2.1316043	7
		35.636745		1.85364118	2.18307038	8

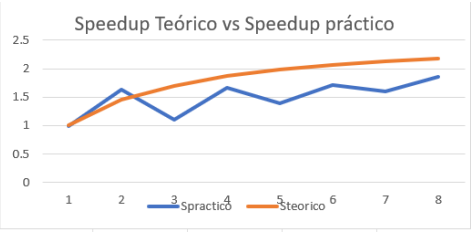

Figura 4: Resultados teóricos y prácticos

4.3.3. Conclusiones

Hemos observado que el Speedup aumenta de forma "parecida.ª lo esperado de forma teórica, hay oscilaciones y valores por encima de lo teórico porque se está corriendo en una máquina con otros programas abiertos y está corriendo el código en un subsistema linux, con las limitaciones que ello conlleva.
Viendo estos resultados prácticos vemos que este programa no tiene una buena escalabilidad fuerte, porque al aumentar el numero de procesadores no aumenta el speedup de forma proporcional.
De forma teórica observamos que éste problema no tiene un buen escalado fuerte porque la función del speedup no tiene una forma lineal, sino una forma que tiende más a ser logarítmica(una asíntota horizontal)
La ley de Amdahl tiene una buena fiabilidad teniendo en cuenta que es una estimación, los valores que da proporciona una idea de como va a ser la escalabilidad fuerte en un futuro, pero no es una ley que haya que tomar de forma muy rigurosa porque el caso práctico nunca se asemeja al teórico.