MEMORIA PROYECTO PRACTICA 2 CPA

Cesar Martínez Chico

Jaime Candel Martínez

28/11/2022

INDICE

INTRODUCCION

En esta práctica hemos trabajado con la interfaz OpenMP para paralelizar un código secuencial, buscando hallar la mejor planificación para conseguir una reducción de su coste temporal sin sacrificar ninguna de sus prestaciones.

EJERCICIO 1

Paralelización del bucle interno de la primera parte: Hemos hecho *private* la variable d para que cada hilo tenga su propia copia, evitando así errores en los accesos a memoria y problemas con la integridad de los datos; también hemos creado una zona critica (duplicando la condición **if** y creando la zona crítica antes del segundo para evitar que hilos que no cumplen la condición se pongan a la cola y saturen la entrada a la zona crítica) para evitar condiciones de carrera a la hora de editar *dmin.*

```
  #pragma omp parallel for private(d)
  for ( off = 1 ; off < w ; off++ ) {
    d  = distance( w-off, &a[3*(y-1)*w], &a[3*(y*w+off)], dmin );
    d += distance( off, &a[3*(y*w-off)], &a[3*y*w], dmin-d );
    // Update minimum distance and corresponding best offset
    if(d < dmin){ //duplicamos el if
        #pragma omp critical
    if ( d < dmin ) { dmin = d; bestoff = off; }
    }
  }
  voff[y] = bestoff;
}
```

Paralelización del bucle de la tercera parte: Hemos creado una zona paralela que comprende toda la parte 3 y puesto la variable v como private para que, igual que en la primera parte, cada hilo tenga su propia v evitando así posibles errores derivados de la computación paralela.

```
  #pragma omp porallel private(v) //v privada
  {
  v = malloc( 3 * max * sizeof(Byte) );
  if ( v == NULL )
    fprintf(stderr,"ERROR: Not enough memory for v\n");
  else {
      #pragma omp for
    for ( y = 1 ; y < h ; y++ ) {
      cyclic_shift( w, &a[3*y*w], voff[y], v );
    }
    free(v);
  }
  }
  free(voff);
}
```

EJERCICIO 2

Comienza de la región paralela y declara la variable v private para que dentro de toda la región cada hilo trabaje con su propia versión de la misma que no varía si otro hilo la manipula.

```
#pragma omp parallel  private(v)
{ //abrimos region paralela
 // Part 1. Find optimal offset of each line with respect to the previous line
```

Paralelización del bucle interno de la primera parte y declara privadas las variables: d, ya que va a ser manipulada y leída por todos los hilos; off, ya que es la referencia a la hora de realizar las iteraciones del bucle y si no fuera privada habría hilos que se saltarían algunas iteraciones; la variable bestoff, que se encuentra dentro de la zona critica; y la variable dmin, que también varios hilos y puede ser una fuente de errores y condiciones de carrera si no se asegura su correcto uso mediante el uso de *pragma omp critical* que solo deja a un único hilo acceder a la variable a la vez.

```
// Part 1. Find optimal offset of each line with respect to the previous line
 #pragma omp parallel for private(d, off, bestoff, dmin)
for ( y = 1 ; y < h ; y++ ) {

  // Find offset of line y that produces the minimum distance between lines y and y
  dmin = distance( w, &a[3*(y-1)*w], &a[3*y*w], INT_MAX ); // offset=0
  bestoff = 0;
  for ( off = 1 ; off < w ; off++ ) {
    d  = distance( w-off, &a[3*(y-1)*w], &a[3*(y*w+off)], dmin );
    d += distance( off, &a[3*(y*w-off)], &a[3*y*w], dmin-d );
    // Update minimum distance and corresponding best offset
     if ( d < dmin ){ // igual que antes duplicamos el if
      #pragma omp critical
          if ( d < dmin ) { dmin = d; bestoff = off; }
     }
  }
  voff[y] = bestoff;
}
```

Como la segunda parte la tiene que ejecutar solo un único hilo se utiliza la instrucción pragma omp parallel single para que solo un hilo la ejecute.

```
// Part 2. Convert offsets from relative to absolute and find maximum offset of any line
#pragma omp single
{ //esta directiva para que la parte 2 QUE NO ES PARALELIZABLE solo la ejecute un unico hilo
max = 0;
voff[0] = 0;
for ( y = 1 ; y < h ; y++ ) {
  voff[y] = ( voff[y-1] + voff[y] ) % w;
  d = voff[y] <= w / 2 ? voff[y] : w - voff[y];
  if ( d > max ) max = d;
}
}
```

Paralelización del bucle de la tercera parte, la más sencilla, ya que no tiene que crear ninguna variable privada que pueda dar problema al ser utilizada en paralelo.

```
,
// Part 3. Shift each line to its place, using auxiliary buffer v
v = malloc( 3 * max * sizeof(Byte) );
if ( v == NULL )
  fprintf(stderr,"ERROR: Not enough memory for v\n");
else {
    #pragma omp for //paralelizamos de igual forma el bucle for
  for ( y = 1 ; y < h ; y++ ) {
    cyclic_shift( w, &a[3*y*w], voff[y], v );
  }
  free(v);
}
}
  free(voff);
}
```

EJERCICIO 3

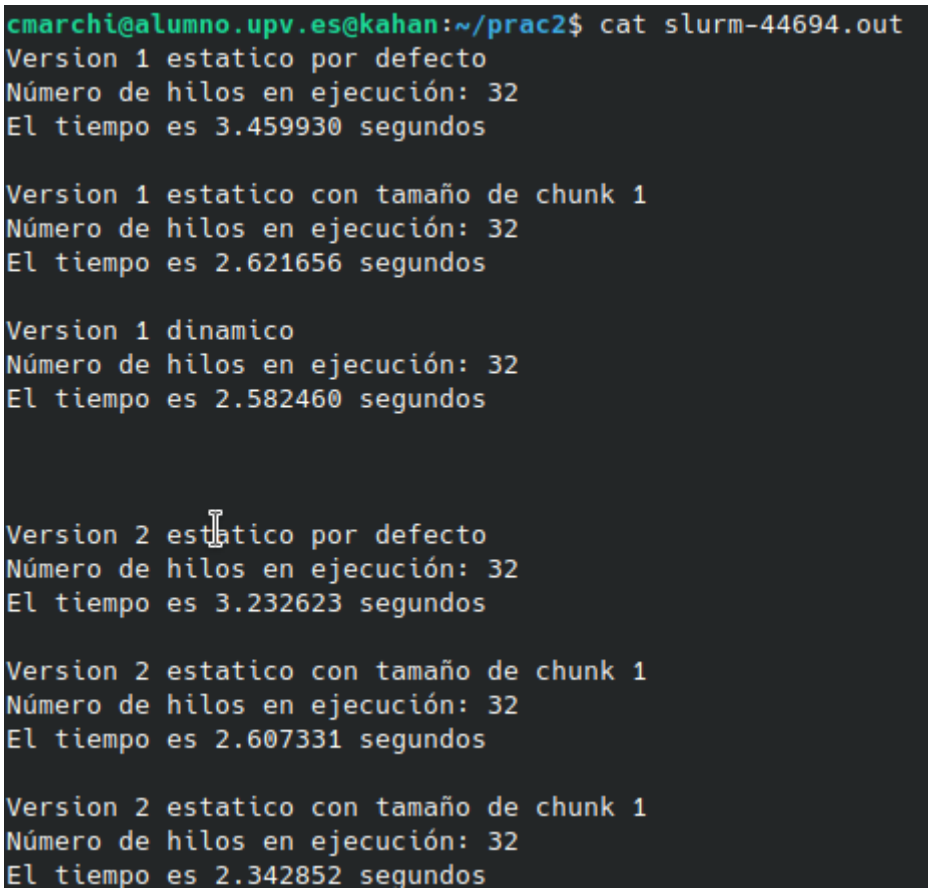
Si eliminamos la condición d<c y el bucle siempre se completa hasta llegar a **n**, la planificación que produciría un mejor equilibrio de carga sería la ***static*** *con tamaño de chunk 1* ya que sabemos que va a ejecutar siempre todo y no va a variar y sabemos que como cada iteración va aumentando en coste computacionalmente es la forma más eficiente de repartirlo, por lo que no hace falta sobrecargar la planificación. Entonces, la peor sería ***dynamic*** *con tamaño de chunk por defecto*, ya que, sobrecargas de manera innecesaria la planificación sabiendo que el bucle siempre se va a completar hasta llegar a **n.**

Por el contrario, si no se hubiera eliminado la condición d<c seria más aconsejable utilizar una planificación **dynamic** con tamaño de chunk por defecto, ya que, en este caso no sabes si el bucle siempre se completará hasta **n**

EJERCICIO 4

Como podemos ver: en la primera versión, la que tiene paralelizado solo el bucle interno de la primera parte y el bucle de la tercera parte, la planificación más eficiente, en cuanto a coste temporal, es la versión *dinámica con tamaño de chunk por defecto*. Sin embargo, en la versión dos es más conveniente la planificación *estática con el tamaño de chunk por defecto*.

En ambos casos se hace uso de un *tamaño de chunk por defecto,* ya que, así el aprovechamiento de la memoria caché es mejor y más eficiente. Sin embargo, en la versión dos es mas conveniente



EJERCICIO 5

No hemos podido hacerlo ya que KAHAN está colapsado en y tenemos que entregar el proyecto ya, entonces, nos es imposible.

hilos	1(secuenc	2	4	8	16	32
tipo						
static defa	61					3,45
static chur	61					2,62
dynamic	61					2,58

version 1

hilos	1(secuenc	2	4	8	16	32
tipo						
static defa	61					3,23
static chur	61					2,6
dynamic	61					2,34

version 2

EJERCICIO 6

Versión privada, donde se hace un reduction a la d lo que hace que ya sea implícitamente privada para cada hilo.

```
int distance( int n, Byte a1[], Byte a2[], int c ) {
  int i, d,e;
  n *= 3; // 3 bytes per pixel (red, green, blue)
  d = 0;
  #pragma omp parallel for private (e) reduction (+:d) //version privada, redcuction implica que sea privada
  for ( i = 0 ; i < n; i++ ) {
      if( d < c ){
    e = (int)a1[i] - a2[i];
    if ( e >= 0 ) d += e; else d -= e;
      } else {break}
     // mi idea es que para mantener la condicion d menor que c y poder paralelizar el bucle,
     //ponerla como una condicion if, que si no se cumple, el hilo abandona el for.
     //asi conseguimos paralelizar el bucle manteniendo la condicion, para no alterar un resultado correcto.
  }
  return d;
}

// Shift line a, of n pixels, cyclically so that pixel a[p] will go to a[0]
// v is an auxiliary array of min(p,n-p) pixels at least
void cyclic_shift( int n, Byte a[], int p, Byte v[] ) {
  int i,d;

  if ( p != 0 ) {
    n *= 3; p *= 3;
    d = n - p;
    // Shift is done from right to left of from left to right
    // depending on which alternative requires less space in the auxiliary
    // array v
    if ( p <= n / 2 ) { // right to left
        #pragma omp parallel for
      for ( i = 0 ; i < p ; i++ ) v[i] = a[i];
      for ( i = p ; i < n ; i++ ) a[i-p] = a[i]; //este bucle NO es paralelizable porque accede a una componente
     //del vector a que otro hilo estará escribiendo, ergo CONDICION DE CARRERA
      #pragma omp parallel for
      for ( i = 0 ; i < p  ; i++ ) a[d+i] = v[i];
    } else { // left to right
       #pragma omp parallel for
      for ( i = 0 ; i < d ; i++ ) v[i] = a[p+i]; // aunque accedamos a p + 1, no hay problema porque al ser solo
      //lectura, ningun hilo a modificar a en este for por tanto no hay condiciones de carrera.
      for ( i = p-1 ; i >= 0 ; i-- ) a[i+d] = a[i]; // no paralelizable por la misma razon que el otro. Un hilo
     //lee lo que otro escribe y viceversa.
     #pragma omp parallel for
      for ( i = 0 ; i < d  ; i++ ) a[i] = v[i];
    }
  }
}
```

Version donde la d es compartida entre los hilos y para solucionarlo se hace critico el acceso a la variable, mediante la declaración de *pragma omp critical* rodeada por la duplicación de las cláusulas condicionales para garantizar que solo entran a la zona critica los hilos que realmente van a poder modificar la variable y evitan así que muchos hilos saturen la entrada a la zona critica cuando no van a poder hacer anda dentro de ella.

```
int distance( int n, Byte a1[], Byte a2[], int c ) {
  int i, d,e;
  n *= 3; // 3 bytes per pixel (red, green, blue)
  d = 0;
  #pragma omp parallel for private (e)  //version compartida
  for ( i = 0 ; i < n; i++ ) {
     if( d < c ){
    e = (int)a1[i] - a2[i];
    if ( e >= 0 ) {
    #pragma omp critical //criticalizo el acceso a d para que pueda ser compartida.
    if ( e >= 0 ) d += e; else d -= e;
    }
      } else {break}
     // mi idea es que para mantener la condicion d menor que c y poder
     //paralelizar el bucle, ponerla como una condicion if, que si no se cumple,
     //el hilo abandona el for. asi conseguimos paralelizar el bucle
     //manteniendo la condicion, para no alterar un resultado correcto.
  }
  return d;
}

// Shift line a, of n pixels, cyclically so that pixel a[p] will go to a[0]
// v is an auxiliary array of min(p,n-p) pixels at least
void cyclic_shift( int n, Byte a[], int p, Byte v[] ) {
  int i,d;

  if ( p != 0 ) {
    n *= 3; p *= 3;
    d = n - p;
    // Shift is done from right to left of from left to right
    // depending on which alternative requires less space in the auxiliary
    // array v
    if ( p <= n / 2 ) { // right to left
         #pragma omp parallel for
      for ( i = 0 ; i < p ; i++ ) v[i] = a[i];
      for ( i = p ; i < n ; i++ ) a[i-p] = a[i]; //este bucle NO es paralelizable porque accede
     //a una componente del vector a que otro hilo estará escribiendo, ergo CONDICION DE CARRERA
     #pragma omp parallel for
      for ( i = 0 ; i < p  ; i++ ) a[d+i] = v[i];
    } else { // left to right
        #pragma omp parallel for
      for ( i = 0 ; i < d ; i++ ) v[i] = a[p+i]; // aunque accedamos a p + 1, no hay problema
     //porque al ser solo lectura, ningun hilo puede modificar a en este for por tanto no hay condiciones de carrera.
      for ( i = p-1 ; i >= 0 ; i-- ) a[i+d] = a[i]; // no paralelizable por la misma razon que el otro.
     //Un hilo lee lo que otro escribe y viceversa.
     #pragma omp parallel for
      for ( i = 0 ; i < d  ; i++ ) a[i] = v[i];
    }
  }
}
```

BIBLIOGRAFIA

- http://algoritmosplanificacion.blogspot.com/2012/08/algoritmos-de-planificacion-estaticos-y.html
- http://algoritmosplanificacion.blogspot.com/2012/08/algoritmos-de-planificacion-estaticos-y.html
- https://es.wikipedia.org/wiki/Chunk
- https://learn.microsoft.com/es-es/cpp/parallel/openmp/3-run-time-library-functions?view=msvc-170