UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Máster Universitario en Computación en la Nube y de Altas Prestaciones

Tecnología de la Programación Paralela (TPP).

Actividades para el aprendizaje de OpenMP

Curso 2022–2023

Instrucciones

Este boletín está formado por una serie de actividades. Es indispensable aportar el código de las actividades 1 a 7 para aprobar. Para cada una de ellas hay que crear una carpeta con nombre "Actividad_X" donde X indica el número de actividad. Dentro de la carpeta figurarán los códigos implementados y los documentos solicitados en cada actividad. Después se creará un fichero comprimido que incluya todas las carpetas y se subirá a la tarea de PoliformaT correspondiente. Borrad todos los ejecutables antes de crear el archivo comprimido.

Actividad 1: Paralelización de bucles en OpenMP

Dado un conjunto de vectores, en esta actividad se va realizar el cálculo de la *media* ($\mu = \frac{1}{n}\sum_{i=1}^{n} v_i$) y la *desviación típica* ($\sqrt{\frac{1}{n}\sum_{i=1}^{n}(v_i - \mu)^2}$) para cada vector v. Para ello, se va a trabajar con el fichero fuente facilitado (`actividad1.c`). Este código genera un conjunto de vectores de diferente tamaño y de valores aleatorios.

Tarea a realizar: En la zona indicada para ello, hay que incluir un código que sea capaz de calcular la *media* y la *desviación típica* de todos los vectores generados. Observad que cada vector esta "apuntado" por un puntero del vector `M`, es decir, `M[v]` es el vector v, para $v = 1, \ldots, n\_vectores$, donde $n\_vectores$ es el número de vectores. Para almacenar la *media* y la *desviación típica* de cada uno de los vectores generados aleatoriamente se han declarado dos vectores dinámicos, `media` y `desvt`, respectivamente, de manera que los valores de la *media* y la *desviación típica* del vector `v` se almacenarán en `media[v]` y `desvt[v]`, respectivamente. El código secuencial debería estar basado en un solo bucle que recorra todos los vectores y, para cada vector, calcule su *media* y su *desviación típica*.

El programa guarda los valores calculados en un fichero. De esta manera se puede comprobar si el resultado es correcto entre diferentes ejecuciones mediante el comando Linux `cmp`.

Una vez realizado el código secuencial hay que paralelizarlo. Esto lo realizaremos siguiendo un "diseño basado en hilos", es decir, mediante paralelización de bucles.

Se puede utilizar la máquina que se desee pero la recomendación es utilizar `knights`. La compilación se puede realizar así:

```
$ gcc -o actividad1 actividad1.c ctimer.c -lm
```

Pruebas a realizar: El programa y su paralelización han de ser estudiadas. Aunque no es necesario realizar una memoria para las actividades, sí es necesario realizar una serie de tablas y/o gráficas con resultados. En este caso, para estudiar la paralelización, se recomienda trabajar con tamaños, por ejemplo, de unos 40000 vectores de un tamaño máximo de 40000. El estudio del algoritmo paralelo debe consistir

en mostrar la evolución de los tiempos cuando el número de cores es de 1, 2, 4, 8, 16 y 24. Además, del tiempo es necesario sacar el incremento de velocidad o *speedup* y la eficiencia para cada caso. Como la paralelización se ha realizado sobre uno o varios bucles, hay que estudiar la diferencia entre una política de planificación de bucles *estática* y otra *dinámica* para todas las combinaciones de número de hilos utilizados. Se puede reducir el estudio a dos o tres tamaños de "`chunk`" en cada caso.

Puntuación: 0.2
Entrega: Como respuesta a esta actividad ha de subirse el código y las tablas o gráficas generadas en el estudio. Si son tablas pueden presentarse en ficheros de texto, o en pdf si se trata de gráficas.

Actividad 2: El juego del Sudoku

El objetivo de esta actividad es implementar un programa paralelo que permita obtener el máximo rendimiento de un entorno paralelo con memoria compartida utilizando un diseño basado en hilos para resolver el conocido juego del **Sudoku**.

Para realizar esta actividad se necesita el siguiente material:

- `sudoku.c`: programa principal.
- `sudoku.h`: archivo de cabecera con la definición de algunas funciones.
- `init_sudoku.c`: función que inicializa el Sudoku a resolver. Existen tres sudoku's predefinidos: muy fácil, normal y muy difícil. Dependiendo del elegido el tiempo de resolución es diferente. Para realizar el estudio de prestaciones se debe utilizar el "muy difícil" mientras que los otros dos se han de utilizar para desarrollo.
- `libsudoku_knights.o`: módulo compilado (binario) con funciones auxiliares (solo funciona en el cluster `knights`)[1].

Para compilar el código secuencial hay que incluir `libsudoku.o` y los dos ficheros `*.c`:

```
$ gcc -fopenmp -o sudoku sudoku.c init_sudoku.c libsudoku_knights.o
```

Descripción del problema: *Backtracking*

Una de las estrategias más conocidas para la resolución de este juego por ordenador es la estrategia de *Backtracking*. Según esta estrategia se representa el tablero del Sudoku mediante una matriz de 9×9. A esta matriz se la nombrará como `sol`. Cada casilla del tablero `sol(i,j)` puede tener una cifra entre 0 y 9, correspondiéndose el 0 con una casilla vacía. Se utiliza una matriz auxiliar de enteros que llamamos `mascara` y cuyos elementos tienen el valor 1 cuando el elemento correspondiente no se puede modificar, ó 0 en caso contrario.

La búsqueda de una solución produce un árbol de posibles soluciones. Para saber si una rama es "prometedora", existirá una función (`es_factible`) que, dadas las coordenadas de una casilla y el tablero, devuelve un 1 si el valor que figura en dicha casilla es una entrada válida, es decir, puede conducir a una solución del Sudoku. Esta función se proporciona. El algoritmo de *Backtracking* se muestra en la Figura 1.

Como puede observarse, la solución es recursiva. Esta aproximación recursiva al algoritmo secuencial la hace más concisa y fácil de entender.

Obsérvese que el tablero está declarado como un vector unidimensional de tamaño 81 aunque, gracias a las macros que han sido definidas al principio, el manejo del tablero se puede realizar de manera más intuitiva, es decir, de la forma `sol( i, j )`.

Sea, por ejemplo, el Sudoku de la Figura 2. El algoritmo de *Backtracking* genera un árbol de soluciones de la siguiente manera. En el primer nivel tenemos un solo nodo raíz que representa el tablero inicial. En el siguiente nivel tenemos todas las posibles soluciones en las cuales el valor de la primera celda vacía es válido. En el Sudoku del ejemplo, los posibles valores de la celda 1×1 son: 1, 2 y 8, lo que da lugar a tres posibles ramas. En el siguiente nivel se generan las posibles soluciones siguiendo el razonamiento anterior tomando el Sudoku representado por el nodo padre y siguiendo el orden de casillas por filas. El

[1] El módulo compilado (binario) `libsudoku.o`: puede utilizarse en el cluster `kahan` si se desea obtener tiempos en ese cluster de manera exclusiva.

```
#define        sol(a,b) sol[(a-1)*9+(b-1)]
#define        mascara(a,b) mascara[(a-1)*9+(b-1)]

void sudoku_sol( int i, int j, int sol[81], int mascara[81] ) {
   int k;
   if( mascara(i, j) == 0 ) {
      for( k = 1; k <= 9; k++ ) {
         sol( i, j ) = k;
         if( es_factible( i, j, sol ) ) {
            if( j < 9 ) {
               sudoku_sol( i, j+1, sol, mascara );
            } else if( i < 9 ) {
               sudoku_sol ( i+1, 1, sol, mascara );
            } else {
               printf("Solucion: \n");
               prin_sudoku(sol);
            }
         }
      }
      sol(i, j) = 0;
   } else {
      if( j < 9 ) {
         sudoku_sol( i , j+1, sol, mascara );
      } else if( i < 9 ) {
         sudoku_sol ( i+1, 1, sol, mascara );
      } else {
         printf("Solucion: \n");
         prin_sudoku(sol);
      }
   }
}
```

Figura 1: Algoritmo secuencial recursivo de *Backtracking* para la resolución de un Sudoku.

árbol de la Figura 3 muestra los primeros niveles del árbol de soluciones. Cada nodo representa la lista de valores asignados a las casillas por orden de filas. Podemos identificar en el algoritmo un bucle `for` que itera sobre la variable `k` y se corresponde con el recorrido de los nodos de un mismo nivel, mientras que cada llamada recursiva a la función `sudoku_sol` se corresponde con el paso a un nodo inferior. El algoritmo, tal como está diseñado, se dice que recorre el árbol de soluciones en *profundidad*.

Aproximación paralela mediante un diseño *basado en hilos*

Una primera idea de asignación de trabajo consiste en asignar cada uno de los nodos del nivel 1 del árbol de soluciones (los tres nodos del nivel 1 del ejemplo de la Figura 3) a un hilo. Evidentemente, con esta aproximación sólo hay trabajo para unos pocos hilos (3 hilos en el ejemplo). Esto lleva a descender en el número de niveles con el objeto de encontrar un nivel en el que exista un número suficiente de tareas que permita distribuir trabajo entre, al menos, un número suficiente de hilos, es decir, al menos tantos como procesadores haya disponibles.

Uno de los inconvenientes de esta aproximación está en que, para descender en el árbol hasta alcanzar un nivel con suficientes nodos, solo trabaja un hilo en dicho descenso perdiéndose oportunidad de paralelismo. Aún cuando coincidan el número de tareas con el de hilos, el problema del desbalanceado de carga continúa debido a que cada rama del árbol puede tener un coste diferente y desconocido a priori. Para conseguir mejor balanceo de la carga es necesario descender a un nivel muy profundo con el objeto de tener más nodos que procesadores, pero esto hace que el hilo inicial tenga que trabajar más tiempo él solo y, por la ley de Amdahl, se pierda oportunidad de paralelismo. Por lo tanto, habrá que buscar una solución de compromiso y ésta será buscada de manera experimental siendo uno de los valores que se deben estudiar.

Para implementar esta opción se sugiere que un hilo recorra los primeros N niveles del árbol de

			7				5	3
	9		3					4
		6		4		8		
					5		9	
4	7			9	6			
9		5		3				
	2	7	5				3	
	4	9	2				8	7
5		3				2	4	

Figura 2: Sudoku inicial de ejemplo.

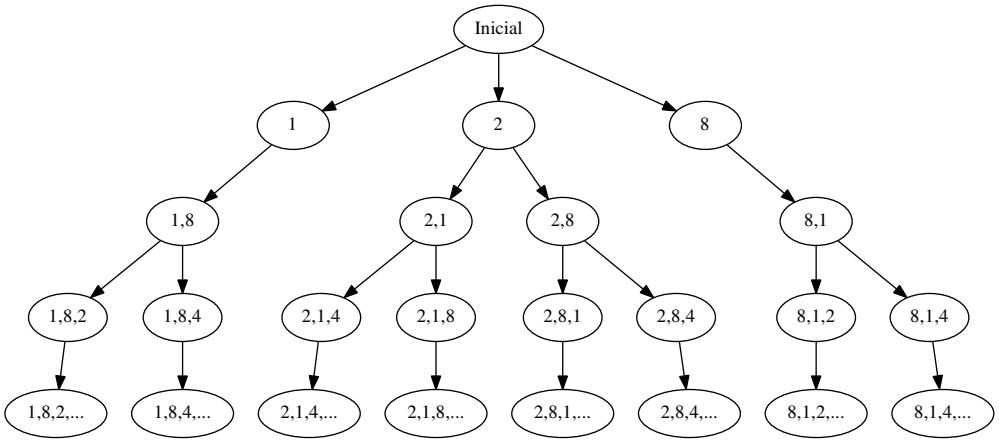

Figura 3: Primeros niveles del árbol de soluciones producido por el Algoritmo de *Backtracking*.

soluciones de manera iterativa para obtener todos los nodos del nivel N. En el algoritmo original, en lugar de llamar a la función `sudoku_sol` (linea 19) se podría introducir el código de la Figura 4.

Según se ve en el código de la Figura 1, un tablero se almacena en un vector de 81 elementos. En el código de la Figura 4 se utilizan dos matrices, `A` y `B`, donde cada fila representa un tablero. Para cada uno de los niveles se recorren todos los nodos (filas de la matriz `A`) desde la primera, `A[0][:]`[2], hasta la `A[tableros-1][:]`, siendo `tableros` el número de nodos en un nivel. Dado el tablero `A[nodo][:]` correspondiente al nodo `nodo`, se generan los nodos hijos, es decir, aquellos que pueden dar lugar a una posible solución. Si es el caso, se almacena el nuevo tablero en `B[j][:]`. Después de recorrer todos los nodos de un nivel se copian los `j` nodos generados en el nivel `nivel` y almacenados en la matriz `B` en la matriz `A` (líneas 21–24) para procesar el siguiente nivel.

Lo anterior es una propuesta basada en una transformación recursivo-iterativa. En realidad, no sería necesario realizar esta transformación. Con algo de habilidad se podría dejar el algoritmo recursivo tal como está con una ligera modificación. Esta modificación consiste en parar la profundización de la recursión en el nivel seleccionado, guardar en ese momento el tablero generado, y regresar en la recursión (*backtracking*). En otras palabras, se desciende en el árbol hasta el nivel deseado utilizando el propio algoritmo recursivo, en lugar de utilizar la transformación anterior (Figura 4).

Ahora, solo queda recorrer mediante un bucle todos los nodos o `tableros` generados almacenados en

[2]Los dos puntos `:` no es notación C, se utiliza para representar todas las componentes de un vector.

```
1   int A[3000][81];
2   int B[3000][81];
3   int nivel, nodo, k, l;
4   for(int l = 0; l < 81; l++) A[0][l] = sol[l];
5   int tableros = 1;
6   for( int nivel = 0; nivel < profundidad; nivel ++){
7     int j = 0;
8     for( int nodo = 0; nodo < tableros; nodo++ ) {
9       int k = 0; while( k < 81 && A[nodo][k] != 0 ) k++;
10      if( k<81 ) {
11        for( int i=1; i<=9; i++ ) {
12          A[nodo][k] = i;
13          if( es_factible( k/9+1, k%9+1, A[nodo] ) ) {
14            for( int l = 0; l<81; l++ ) { B[j][l] = A[nodo][l]; }
15            j++;
16          }
17          A[nodo][k] = 0;
18        }
19      }
20    }
21    tableros = j;
22    for( int i = 0; i<tableros; i++ )
23      for( int k = 0; k<81; k++ )
24        A[i][k] = B[i][k];
25  }
26
27  for(int tablero = 0; tablero < tableros; tablero++) {
28    int mascara[81];
29    for ( int i = 0; i < 81; i++ ) mascara[i] = A[tablero][i] != 0;
30    sudoku_sol(1,1,A[tablero],mascara);
31  }
```

Figura 4: Generación de los nodos del nivel N de un árbol de soluciones para el problema del Sudoku

la matriz `A` y llamar a la función recursiva (`sudoku_sol`) para cada uno de ellos (lineas 27-31). Se ve claramente que este bucle, cuyas iteraciones representan ramas, es paralelizable.

Tarea a realizar

La tarea realizar es la **paralelización *basada en hilos* del Sudoku**:

- Compilar el código secuencial siguiente:

```
$ gcc -fopenmp -o sudoku_estatico sudoku_estatico.c init_sudoku.c libsudoku_knights.o \
      ctimer.c
```

- El programa debe leer por linea de comandos un número entero que indique la `profundidad`.
- Paralelizad el programa y ejecutadlo siempre con el máximo número de hilos disponible variando el nivel de profundidad.
- Probad diferentes políticas de planificación del bucle, unas estáticas y otras dinámicas para los diferentes niveles de profundidad seleccionados en el apartado anterior. Generar una tabla con los datos temporales y número total de tableros generados en función de la profundidad. (No es necesario obtener el incremento de velocidad y la eficiencia.) **Nota**: Utilizar el Sudoku `"muy dificil"` para obtener tiempos.

Algoritmo 1 Algoritmo escalar para la descomposición de Cholesky de una matriz A simétrica y definida positiva.

```
1: function CHOL_ESCALAR( A ) return C
2:     C ← A                                  ▷ Se copia la matriz de entrada A en la de salida C
3:     for k = 1 → n do
4:         c = √C(k,k)
5:         C(k,k) = c
6:         for i = k+1 → n do
7:             C(i,k) = C(i,k)/c
8:         end for
9:         for i = k+1 → n do
10:            for j = k+1 → i−1 do
11:                C(i,j) = C(i,j) − C(i,k) · C(j,k)
12:            end for
13:            C(i,i) = C(i,i) − C(i,k) · C(i,k)
14:        end for
15:    end for
16: end function
```

Puntuación: 0.3
Entrega: Como respuesta a esta actividad ha de subirse el código y las tablas o gráficas generadas en el estudio.

Actividad 3: Descomposición de Cholesky

Dada una matriz $A \in \mathcal{R}^{n\times n}$ *simétrica* y *definida positiva*, la descomposición de Cholesky consiste en encontrar un factor C tal que:

$$A = C \cdot C^T \ ,$$

siendo $C \in \mathcal{R}^{n\times n}$ triangular inferior.

El algoritmo para la resolución de este problema es el Algoritmo 1 (CHOL_ESCALAR). Como se puede observar, este algoritmo solo referencia la parte triangular inferior de la matriz. La matriz A debe ser *definida positiva*. Una matriz *definida positiva* cumple que todos sus valores propios son mayores que cero. Esta característica asegura que la operación de raíz cuadrada (linea 4) se puede realizar, es decir, que el radicando no sea negativo.

Con objeto de mejorar la eficiencia del algoritmo de descomposición de Cholesky, se propone a continuación un algoritmo *por bloques* que resuelve el mismo problema (Algoritmo 2 — CHOL_BLOQUES). Esta versión reduce el tiempo de ejecución en secuencial debido al mejor aprovechamiento de las memorias caché, pero también puede mejorar la eficiencia de la implementación paralela del mismo. La entrada y la salida del algoritmo son las mismas con la diferencia de que ahora tenemos un parámetro más de entrada: el tamaño de bloque b. Este parámetro puede variarse sin que ésto afecte al resultado[3]. La variación de b permitirá obtener tiempos distintos de ejecución que vienen determinados por el tamaño de la caché entre otros factores. El mejor valor para b dependerá de la máquina en la que se trabaje y será obtenido experimentalmente.

De la misma manera que en el algoritmo escalar, este algoritmo solo referencia la parte triangular inferior de la matriz A. Los bucles de tipo $k = 1 : b : n$ significan que el índice toma los valores $k = 1, b+1, b+2, \ldots$ siempre y cuando $k \leq n$. Este bucle corresponde a un bucle en C del tipo

```
for( k = 1; k <= n; k += b ) {
  . . .
}
```

El Algoritmo 2 es esencialmente el algoritmo escalar donde las operaciones con escalares son ahora operaciones con **bloques cuadrados** de matrices.

[3] Al menos en precisión infinita.

Algoritmo 2 Algoritmo por bloques para la descomposición de Cholesky de una matriz A simétrica y definida positiva.

```
1: function CHOL_BLOQUES( A, b ) return C
2:     C ← A                                   ▷ Se copia la matriz de entrada A en la de salida C
3:     for k = 1 : b : n do
4:         m_k = mín(k + b − 1, n)
5:         D = CHOL_ESCALAR(C(k : m_k, k : m_k))                                   ▷ POTRF
6:         C(k : m_k, k : m_k) = D
7:         for i = k + b : b : n do
8:             m_i = mín(i + b − 1, n)
9:             C(i : m_i, k : m_k) ← C(i : m_i, k : m_k)/D^T                       ▷ TRSM
10:        end for
11:        for i = k + b : b : n do
12:            m_i = mín(i + b − 1, n)
13:            for j = k + b : b : i − 1 do
14:                m_j = min(j + b − 1, n)
15:                C(i : m_i, j : m_j) ← C(i : m_i, j : m_j) − C(i : m_i, k : m_k) · C(j : m_j, k : m_k)^T   ▷ GEMM
16:            end for
17:            C(i : m_i, i : m_i) ← C(i : m_i, i : m_i) − C(i : m_i, k : m_k) · C(i : m_i, k : m_k)^T        ▷ SYRK
18:        end for
19:    end for
20: end function
```

Las operaciones etiquetadas como POTRF, TRSM, GEMM y SYRK en el algoritmo referencian a las rutinas de las librerías BLAS/LAPACK. Estas rutinas son:

- DPOTRF calcula la factorización de Cholesky de una matriz real de doble precisión simétrica y definida positiva. La cabecera de esta rutina así como la descripción de sus parámetros se pueden encontrar en `http://www.netlib.org/clapack/clapack-3.2.1-CMAKE/SRC/VARIANTS/cholesky/TOP/dpotrf.c`, o `https://github.com/vtjnash/atlas-3.10.0/blob/master/interfaces/lapack/C/src/clapack_dpotrf.c` para la interfaz C.
- DTRSM resuelve un sistema de ecuaciones con múltiples vectores independientes. La cabecera de esta rutina así como la descripción de sus parámetros se pueden encontrar en `http://www.netlib.org/clapack/cblas/dtrsm.c`, o `https://software.intel.com/es-es/node/520783` para la interfaz C.
- DGEMM multiplica dos matrices. La cabecera de esta rutina así como la descripción de sus parámetros se pueden encontrar en `http://www.netlib.org/clapack/cblas/dgemm.c`, o `https://software.intel.com/es-es/node/520775` para la interfaz C.
- DSYRK realiza una actualización de rango k sobre una matriz simétrica de la forma

  $$C \leftarrow \beta C + \alpha A^T \cdot A \,,$$

  donde α y β son escalares, C es una matriz simétrica cuadrada de orden n y A es una matriz de tamaño $n \times k$. La cabecera de esta rutina así como la descripción de sus parámetros se pueden encontrar en `http://www.netlib.org/clapack/cblas/dsyrk.c`, o `https://software.intel.com/es-es/node/520780` para la interfaz C. Como se puede ver en la descripción, solo la parte triangular inferior o superior de C es referenciada.

Para realizar las tareas siguientes se dispone de un código secuencial de partida: `cholesky.c`. Para compilar los algoritmos del Cholesky utilizad la siguiente línea de compilación:

```
$ gcc -o cholesky cholesky.c -llapack -lblas -lm ctimer.c
```

(No haced caso de los *warnings*).

Observad que en el lugar del código donde se indica `LLAMADA LA FUNCIÓN PRINCIPAL` existe la posibilidad de llamar a tres códigos diferentes:

- `info = cholesky_escalar( n, A );`: Algoritmo escalar de Cholesky.
- `info = cholesky_bloques( n, b, A );`: Algoritmo por bloques de Cholesky.
- `dpotrf_( "L", &n, A, &n, &info );` : Algoritmo de Cholesky implementado en la biblioteca LAPACK.

Dependiendo de cuál de estas tres funciones esté *descomentada* se ejecutará una u otra y, por tanto, el tiempo que se obtendrá será el tiempo de ejecución de la misma. El error deberá ser pequeño ($\approx 10^{-11}$) como indicativo de que el resultado es correcto.

Tarea a realizar

Estudio secuencial:

1. Realizar un estudio del tamaño de bloque en el algoritmo de Cholesky por bloques. Tomad tres valores de tamaño de problema, variad el tamaño de bloque en un rango lo suficientemente amplio como para encontrar un tamaño de bloque en el cual el tiempo sea mínimo para los tres tamaños de problema. Todo ello en secuencial.
2. Realizar una comparación entre Cholesky escalar y por bloques en secuencial para diferentes tamaños de problema.

Estudio paralelo:

1. Realizar una implementación paralela del algoritmo de Cholesky utilizando un **diseño basado en hilos** (paralelización de bucles) de la función `cholesky_bloques`. *Por comodidad se han declarado unas macros al principio del programa que permiten referenciar el elemento i, j de una matriz C como `C(i,j)` en lugar de la forma tradicional de `C`.*
2. Con un tamaño de bloque fijo (el mejor) realizar un estudio experimental de Cholesky variando el tamaño de problema y el número de hilos. Obtened tiempo de ejecución, *speedup* y eficiencia.

Puntuación: 0.3
Entrega: Como respuesta a esta actividad ha de subirse el código y las tablas o gráficas generadas en el estudio.

Actividad 4: Paralelización de bucles en OpenMP con tareas

Implementad el código implementado en la Actividad 1 con un **diseño basado en tareas**, es decir, utilizando la directiva `task` de OpenMP. Realizad exactamente el mismo estudio que el efectuado en aquella actividad para comparar ambos enfoques de paralelización con números.

Puntuación: 0.2
Entrega: Como respuesta a esta actividad ha de subirse el código y las tablas o gráficas generadas en el estudio.

Actividad 5: Paralelización de algoritmos recursivos en OpenMP

Esta actividad consiste en paralelizar con tareas de OpenMP dos algoritmos recursivos de ordenación. La clave está en saber cuándo es necesario sincronizar las tareas con la directiva `taskwait`.

1. Algoritmo de *Quicksort*: Implementa una versión eficiente en OpenMP para resolver el problema de la ordenación de un vector mediante el método del *quicksort*. Existe un código en la carpeta `material` para realizar esta actividad. Solo se requiere aportar el código implementado para esta actividad.
2. Algoritmo de *Mergesort*: Implementa una versión eficiente en OpenMP para resolver el problema de la ordenación de un vector mediante el método del *mergesort*. Existe un código en la carpeta `material` para realizar esta actividad. Solo se requiere aportar el código implementado para esta actividad.

Puntuación: 0.1
Entrega: Como respuesta a esta actividad ha de subirse exclusivamente el código.

Actividad 6: Paralelización del Sudoku basada en tareas

La tarea a realizar consiste en volver al ejemplo del Sudoku (Actividad 2) y realizar una paralelización del mismo **basada en tareas**.

- Para realizar esta tarea es necesario partir del código secuencial inicial que resuelve el Sudoku. A este código hay que añadir aquellas directivas de OpenMP relacionadas con tareas (`task`). La estrategia de diseño es aquella utilizada en algoritmos recursivos como *Quicksort* o *Mergesort*.
- Una vez realizada la implementación del algoritmo, hay que realizar una comparación con la versión basada en hilos implementada en la Actividad 2. En este caso, no existe `profundidad` ni tampoco un bucle para el que variar la política de planificación, luego, en este caso, solo tenemos un tiempo de ejecución paralelo. Realizar una comparativa del tiempo de ejecución con 1, 2, 4, 8, 16 y 24 hilos con el mejor algoritmo paralelo obtenido en la Actividad 2.

Puntuación: 0.2
Entrega: Como respuesta a esta actividad ha de subirse el código y las tablas o gráficas generadas en el estudio.

Actividad 7: Paralelización de Cholesky basada en tareas

- Realizar una implementación paralela del algoritmo de Cholesky con OpenMP utilizando un **diseño basado en tareas**. Esto consiste en identificar aquellas tareas que pueden ser ejecutadas concurrentemente y crearlas mediante la directiva `task`. Las tareas pueden ser identificadas con las llamadas a las funciones de BLAS/LAPACK. Hay que observar cuidadosamente si es necesario sincronizar las tareas o no (`taskwait`).
- Realizar una implementación paralela del algoritmo de Cholesky con OpenMP utilizando un **diseño basado en tareas** con dependencias (cláusula `depend`).
- **Tarea a realizar**: Realizar una comparativa entre las tres versiones de Cholesky por bloques:
  - diseño basado en hilos (Actividad 3),
  - diseño basado en tareas (sin dependencias),
  - diseño basado en tareas con dependencias.

  Los resultados pueden ofrecerse en forma de tablas pero será más claro si se muestran en figuras. Se sugiere, por ejemplo, crear una figura por tamaño de problema con tres gráficas, correspondientes a cada uno de los enfoques, que varíen con el número de hilos utilizado (1, 2, 4, 8, 16, 24). Pueden mostrarse los resultados de otras maneras diferentes.

Puntuación: 0.3
Entrega: Como respuesta a esta actividad ha de subirse el código y las tablas o gráficas generadas en el estudio.