UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Máster Universitario en Computación en la Nube y de Altas Prestaciones

Tecnología de la Programación Paralela (TPP).

Ejercicios OpenMP Sesión 2

Ejercicio 1: 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_gpu.o`: módulo compilado (binario) con funciones auxiliares (solo funciona en el servidor `gpu`).

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

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

```
#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.

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 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

			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.

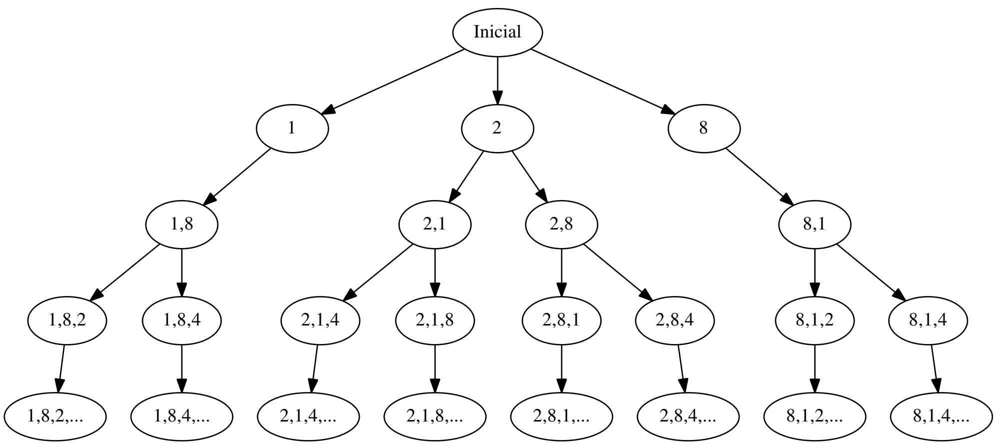

Figura 3: Primeros niveles del árbol de soluciones producido por el Algoritmo de *Backtracking*.

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 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][:]`[1], 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 re-

[1]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

cursión 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 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_gpu.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

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: $\quad C \leftarrow A$ $\triangleright$ Se copia la matriz de entrada A en la de salida C
3: $\quad$ **for** $k = 1 \to n$ **do**
4: $\quad\quad c = \sqrt{C(k,k)}$
5: $\quad\quad C(k,k) = c$
6: $\quad\quad$ **for** $i = k+1 \to n$ **do**
7: $\quad\quad\quad C(i,k) = C(i,k)/c$
8: $\quad\quad$ **end for**
9: $\quad\quad$ **for** $i = k+1 \to n$ **do**
10: $\quad\quad\quad$ **for** $j = k+1 \to i-1$ **do**
11: $\quad\quad\quad\quad C(i,j) = C(i,j) - C(i,k) \cdot C(j,k)$
12: $\quad\quad\quad$ **end for**
13: $\quad\quad\quad C(i,i) = C(i,i) - C(i,k) \cdot C(i,k)$
14: $\quad\quad$ **end for**
15: $\quad$ **end for**
16: **end function**

obtener el incremento de velocidad y la eficiencia.) **Nota**: Utilizar el Sudoku `"muy dificil"` para obtener tiempos.

Ejercicio 2: Descomposición de Cholesky

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

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

siendo $C \in \mathbb{R}^{n\times n}$ 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[2]. 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.

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

[2] Al menos en precisión infinita.

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

function CHOL_BLOQUES(A, b) **return** C
    $C \leftarrow A$ ▷ Se copia la matriz de entrada A en la de salida C
    for $k = 1 : b : n$ **do**
        $m_k = \text{mín}(k + b - 1, n)$
        $D =$ CHOL_ESCALAR$(C(k : m_k, k : m_k))$ ▷ POTRF
        $C(k : m_k, k : m_k) = D$
        for $i = k + b : b : n$ **do**
            $m_i = \text{mín}(i + b - 1, n)$
            $C(i : m_i, k : m_k) \leftarrow C(i : m_i, k : m_k)/D^T$ ▷ TRSM
        end for
        for $i = k + b : b : n$ **do**
            $m_i = \text{mín}(i + b - 1, n)$
            for $j = k + b : b : i - 1$ **do**
                $m_j = min(j + b - 1, n)$
                $C(i : m_i, j : m_j) \leftarrow C(i : m_i, j : m_j) - C(i : m_i, k : m_k) \cdot C(j : m_j, k : m_k)^T$ ▷ GEMM
            end for
            $C(i : m_i, i : m_i) \leftarrow C(i : m_i, i : m_i) - C(i : m_i, k : m_k) \cdot C(i : m_i, k : m_k)^T$ ▷ SYRK
        end for
    end for
end function

- 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`.

- 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`.

- 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`.

- 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`. 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 A 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.

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 obtenido en el estudio secuencial) realizar un estudio experimental de Cholesky variando el tamaño de problema y el número de hilos. Obtened tiempo de ejecución, *speedup* y eficiencia.