Introducción a la Programación Paralela - OpenMP

Dado el siguiente código ya paralelizado con OpenMP, que corresponde al conocido problema de Fibonacci en su versión recursiva:

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

// The n-result in the fibonacci sequence
int fib(int n) {
   int i, j;
   if (n<2)
       return n;
   else {
#pragma omp task shared(i)
       i=fib(n-1);
#pragma omp task shared(j)
       j=fib(n-2);
#pragma omp taskwait
       return i+j;
   }
}

int main(int argc, char **argv){
   int n, result;
   char *a = argv[1];
   n = atoi(a);

#pragma omp parallel
   {
#pragma omp single
       result = fib(n);
   }
   printf("Result is %d\n", result);
}
```

Poné a funcionar en tu computadora el código, previo a instalar OpenMP (ver videos en el material del curso).
Este código utiliza las construcciones task y taskwait, que no vimos en las slides. A no desesperar! La idea es que investigues cómo funcionan acá:
https://hpc2n.github.io/Task-based-parallelism/branch/master/task-basics-1
Luego, respondé a las siguientes preguntas:

a) ¿Por qué crees que se ha utilizado cada #pragma en esos lugares específicos? Explicá qué sentido tiene esta combinación para paralelizar el programa.

b) Utilizá el comando de consola "time", para comprobar si el tiempo del programa completo se reduce al pasar de la versión compilada secuencial (desactivando OpenMP al compilar) a la que utiliza OpenMP con varios hilos. Podés también utilizar omp_get_wtime() para medir por separado el tiempo total y el de la parte paralela. Parametrizar el programa con un valor por encima de 45 debiera llevar a tiempos razonablemente altos para probar. ¿Qué sucede y por qué?

c) Implementá las posibles optimizaciones clásicas para problemas recursivos en caso que apliquen (transformación a código iterativo, memoization, thresholding). Luego, siguiendo la lógica de b), reimplementá lo necesario y probá el efecto de las optimizaciones midiendo nuevamente los tiempos. ¿Qué cambios hay?

Ahora, dado el siguiente código secuencial (sin OpenMP) que describe el problema de las reinas:

```c
#include <stdio.h>
#include <stdbool.h>

#define N 4

long long solutions = 0;

bool is_safe(int *board, int row, int col) {
    for (int i = 0; i < row; i++) {
        if (board[i] == col ||
          board[i] - i == col - row ||
          board[i] + i == col + row)
            return false;
    }
    return true;
}

void solve(int row, int *board) {
    if (row == N) {
        solutions++;
        return;
    }

    for (int col = 0; col < N; col++) {
        if (is_safe(board, row, col)) {
            board[row] = col;
            solve(row + 1, board);
        }
    }
}



int main() {
    int board[N];
    solve(0, board);
    printf("Soluciones para %d reinas: %lld\n", N, solutions);
    return 0;
}
```

d) Parametrizá el programa para que reciba por argumento de línea de comando el tamaño del tablero. Luego, pensá una solución con OpenMP utilizando las directivas que prefieras. Podés usar las del programa anterior de Fibonacci o cualquiera vista en teoría.
e) Como en el caso de b), compará el tiempo del programa secuencial y el paralelo para distintos tamaños de tableros. Podés también utilizar distinto número de hilos si preferís. ¿Qué comportamiento observás?
f) Al igual que c), podés implementar una variante optimizada. Te sugiero thresholding. ¿Ves algún efecto positivo?
g) Elaborá una conclusión acerca de lo realizado, tanto en codificación como en experimentación. ¿Qué podés a partir de la aplicación de OpenMP en ambos códigos? ¿Fue beneficioso OpenMP para ambos casos, y por qué?