Evidencia II

Informe Sobre Implementación y Comparaciones

Docente: Victor Eduardo Martinez Abaunza

Carrera: Ingeniería de Software

Curso: Computación Paralela y Distribuida

Discente:

Leandro Igor Estrada Santos,
lestradas@ulasalle.edu.pe

Facultad de Ingeniería, Universidad La Salle - Arequipa

Índice

1. Ejercicio 1

- Mencione un mecanismo de sincronización que puede ser realizado con Open MP y presente un ejemplo de dicho mecanismo.

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

#define NUM_EJECUCIONES 10
int main() {
    int num_iteraciones = 900000;
    int num_hilos = 8;
    double tiempos_sin[NUM_EJECUCIONES];
    double tiempos_con[NUM_EJECUCIONES];
    double suma_tiempo_sin = 0.0, suma_tiempo_con = 0.0;

    for (int ejecucion = 0; ejecucion < NUM_EJECUCIONES; ejecucion++) {
        int suma_sin_sincronizacion = 0;
        int suma_con_sincronizacion = 0;

        double tiempo_inicio_sin = omp_get_wtime();

        #pragma omp parallel num_threads(num_hilos)
        {
            for (int i = 0; i < num_iteraciones; i++) {
                #pragma omp atomic
                suma_sin_sincronizacion += 1;
            }
        }
        double tiempo_final_sin = omp_get_wtime();
        double tiempo_inicio_con = omp_get_wtime();

        #pragma omp parallel num_threads(num_hilos)
        {
            for (int i = 0; i < num_iteraciones; i++) {
                #pragma omp critical
                {
                    suma_con_sincronizacion += 1;
                }
            }
        }
        double tiempo_final_con = omp_get_wtime();

        tiempos_sin[ejecucion] = tiempo_final_sin - tiempo_inicio_sin;
        tiempos_con[ejecucion] = tiempo_final_con - tiempo_inicio_con;

        suma_tiempo_sin += tiempos_sin[ejecucion];
        suma_tiempo_con += tiempos_con[ejecucion];

```

```
45          printf("\nEjecucion %d:\n", ejecucion + 1);
46          printf("Suma sin sincronizacion: %d\n", suma_sin_sincronizacion)
                ;
47          printf("Tiempo: sin sincronizacion: %.4f segundos\n",
                tiempos_sin[ejecucion]);
48          printf("Suma con sincronizacion: %d\n", suma_con_sincronizacion)
                ;
49          printf("Tiempo: con sincronizacion: %.4f segundos\n",
                tiempos_con[ejecucion]);
50      }
51
52      double promedio_tiempo_sin = suma_tiempo_sin / NUM_EJECUCIONES;
53      double promedio_tiempo_con = suma_tiempo_con / NUM_EJECUCIONES;
54
55      printf("\nPromedio de tiempos de ejecucion:\n");
56      printf("Promedio sin sincronizacion: %.4f segundos\n",
            promedio_tiempo_sin);
57      printf("Promedio con sincronizacion: %.4f segundos\n",
            promedio_tiempo_con);
58
59      return 0;
60  }
61      \
```

- El código presentado demuestra el uso de dos mecanismos de sincronización provistos por OpenMP: atomic y critical. Ambos son utilizados para asegurar que múltiples hilos puedan acceder y modificar una variable compartida sin generar condiciones de carrera.
  - Análisis de Resultados:
    - En cada ejecución del ciclo principal, el código realiza ambas secciones (sin y con sincronización) y almacena los tiempos de ejecución. Al final, se calcula un promedio de tiempos para evaluar el rendimiento.
    - Se espera que la sección con la directiva atomic tenga un mejor rendimiento en comparación con critical, dado que las operaciones atómicas son menos costosas en términos de bloqueo que las secciones críticas, que tienen una mayor sobrecarga.
  - Conclusión sobre el Mecanismo de Sincronización:
    - El código muestra cómo OpenMP ofrece múltiples mecanismos de sincronización, siendo atomic más eficiente para operaciones simples como incrementos, mientras que critical es más flexible pero más costoso. La elección del mecanismo adecuado depende de la complejidad de las operaciones a realizar y el nivel de precisión requerido en el control de acceso concurrente.

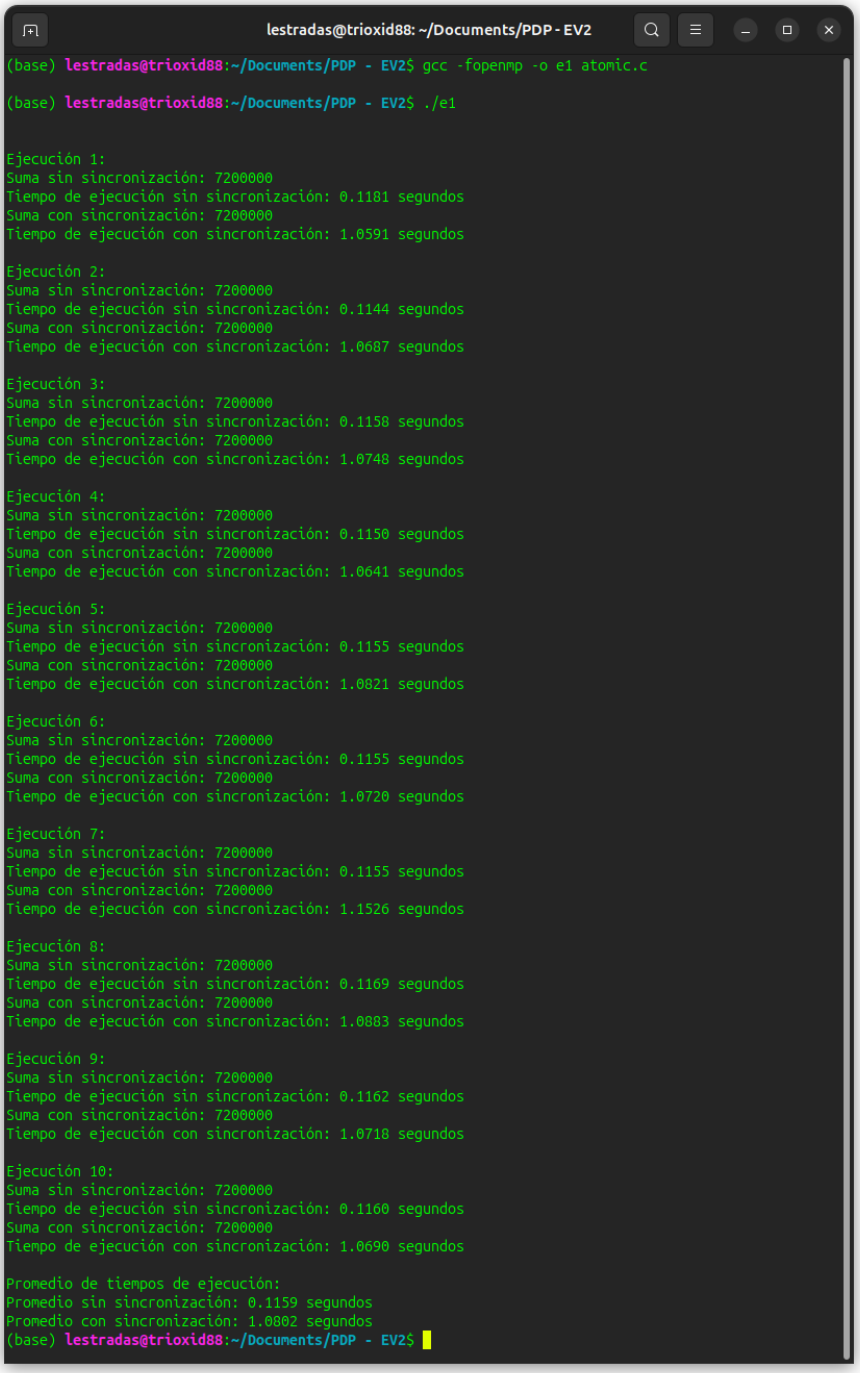

Figura 1: Resultados de la ejecución.

2. Ejercicio 2

- Utilice los códigos para calcular el valor de Pi (Leibinz y Montecarlo).
  - En términos de iteraciones ¿Cuál seria el mejor algoritmo para tener una mejor precisión del cálculo con el menor número de iteraciones?
  - De acuerdo con el número de núcleos disponibles en su máquina virtual ¿Cuál es el número máximo de hilos que pueden ser utilizados para que el tiempo de ejecución disminuya?

,

- Cálculo de Pi usando Leibniz

```
#include <stdio.h>
#include <time.h>
#include "omp.h"

int main() {
    int numeroHilos, idHilo;
    clock_t tiempo_inicio, tiempo_final;
    numeroHilos = omp_get_max_threads();
    omp_set_num_threads(numeroHilos);
    double respuesta = 0.0, sumasParciales[numeroHilos];
    long numeroIteraciones;

    printf("Ejecucion con %d threads:\n", numeroHilos);
    printf("Ingresar el numero de iteraciones: ");
    scanf("%ld", &numeroIteraciones);

    tiempo_inicio = clock();

    #pragma omp parallel private(idHilo) shared(sumasParciales)
    {
        int idHilo = omp_get_thread_num();
        sumasParciales[idHilo] = 0.0;
        for (long indice = idHilo; indice < numeroIteraciones;
           indice += numeroHilos) {
            if (indice % 2 == 0) {
                sumasParciales[idHilo] += 4.0 / (2.0 * indice + 1.0)
                    ;
            } else {
                sumasParciales[idHilo] -= 4.0 / (2.0 * indice + 1.0)
                    ;
            }
        }
    }

```

```
    tiempo_final = clock();

    for (int indice = 0; indice < numeroHilos; indice++) {
        respuesta += sumasParciales[indice];
    }

    printf("La respuesta es: %.8f\n", respuesta);
    printf("Tiempo de ejecucion: %f segundos\n", (double)(
        tiempo_final - tiempo_inicio)/CLOCKS_PER_SEC);
    return 0;
}
```

- Cálculo de Pi usando Montecarlo (Comprende ligeras modificaciones debido a ciertos errores en la versión original de la tarea.)

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

int main(int argc, char *argv[]) {
    int i, count;  // Puntos dentro del cuarto de circulo unitario
    unsigned short xi[3];  // Semilla para numeros aleatorios
    int samples;  // Numero de puntos a generar
    double x, y;  // Coordenadas de los puntos
    double pi;    // Estimacion de Pi

    samples = atoi(argv[1]);

    #pragma omp parallel
    {
        xi[0] = 1;  // Configurar semilla de numeros aleatorios
        xi[1] = 1;
        xi[2] = omp_get_thread_num();
        count = 0;

        #pragma omp for firstprivate(xi) private(x, y) reduction(+:
            count)
        for (i = 0; i < samples; i++) {
            x = erand48(xi);
            y = erand48(xi);
            if (x * x + y * y <= 1.0) {
                count++;
            }
        }
    }

    pi = 4.0 * (double)count / (double)samples;
    printf("Count = %d, Samples = %d, Estimate of pi: %7.5f\n",
        count, samples, pi);
}
```

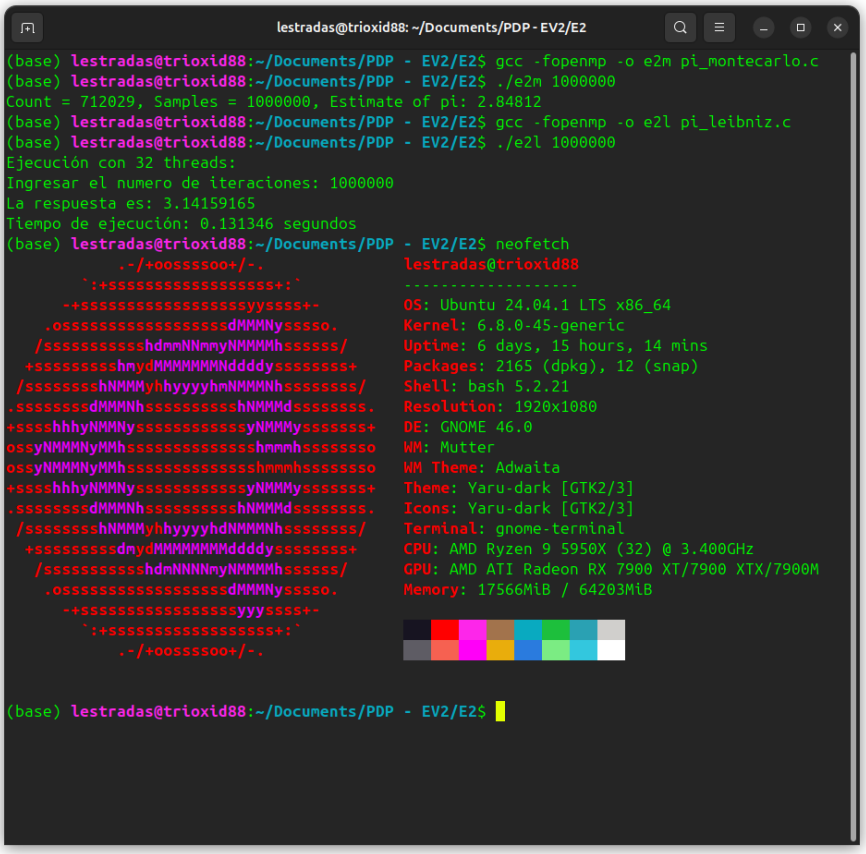

Figura 2: Resultados de la ejecución y especificaciones del hardware.

-

- Análisis:

  1. Comparación de los algoritmos en términos de iteraciones

     a) Leibniz:

     - Este algoritmo tiene una convergencia lenta. Para obtener una precisión aceptable del valor de Pi, se necesitan millones de iteraciones. En pocas iteraciones, el resultado puede estar muy alejado de Pi.
     - El principal problema de Leibniz es que, aunque su implementación es simple, la convergencia es demasiado lenta y no mejora significativamente con la paralelización.

     b) Montecarlo:

     - El método de Montecarlo es un algoritmo estocástico que puede aproximar Pi con un número moderado de iteraciones. Aunque no garantiza una precisión exacta con menos iteraciones, es más eficiente en escenarios de alto paralelismo.
     - La precisión de Montecarlo mejora cuando el número de muestras es muy alto. Con pocas iteraciones, puede producir estimaciones inexactas debido a la naturaleza aleatoria de los puntos generados.

  2. ¿Cuál sería el mejor algoritmo para obtener una mejor precisión con el menor número de iteraciones?

     - En términos de pocas iteraciones, el algoritmo de Leibniz es mejor, ya que, aunque converge lentamente, ofrece una precisión aceptable en sus primeras iteraciones.
     - En términos de muchas iteraciones, el algoritmo de Montecarlo tiende a ofrecer mejores resultados, ya que es más adecuado para aprovechar la paralelización y puede alcanzar una precisión similar con un número menor de iteraciones en comparación con Leibniz.

3. Conclusiones:

   - Algoritmo de Leibniz: Es más preciso en pocas iteraciones, pero tiene una convergencia muy lenta. Requiere millones de iteraciones para obtener una precisión adecuada de Pi.
   - Algoritmo de Montecarlo: Ofrece una aproximación rápida y efectiva con paralelización, pero requiere un alto número de muestras para mejorar la precisión. Es más adecuado para ser paralelizado eficientemente.
   - Número de hilos:
     - Usar 32 hilos es óptimo para ambos algoritmos respecto a mi pc, ya que esto aprovecha completamente los recursos de hardware. Para el algoritmo de Montecarlo, la paralelización proporciona una mayor ganancia en términos de tiempo de ejecución, mientras que en el caso de Leibniz, aunque también se acelera el cálculo, la mejora es menos significativa debido a su lenta convergencia.
     - No es recomendable utilizar más de 32 hilos, ya que no obtendría beneficios adicionales y podría incluso perjudicar el rendimiento debido a la sobrecarga de gestión de hilos.

3. Ejercicio 3

- Fibonacci con Sections modificado:

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

long fibonacci(long numero) {
    if (numero == 1 || numero == 2) {
        return 1;
    } else {
        return fibonacci(numero - 1) + fibonacci(numero - 2);
    }
}
int main() {
    double tiempo_inicio, tiempo_final;
    int numeroHilos = omp_get_max_threads();
    omp_set_num_threads(numeroHilos);
    long respuesta = 0, numero;
    printf("Ingresar un numero: ");
    scanf("%ld", &numero);
    tiempo_inicio = omp_get_wtime();  // Iniciar cronometro

    #pragma omp parallel sections
    {
        #pragma omp section
        {
            long subrespuesta = fibonacci(numero - 2);
            printf("El hilo %d descubrio que fibonacci(%ld) = %ld\n",
                omp_get_thread_num(), numero - 2, subrespuesta);
            #pragma omp atomic
            respuesta += subrespuesta;
        }
        #pragma omp section
        {
            long subrespuesta = fibonacci(numero - 1);
            printf("El hilo %d descubrio que fibonacci(%ld) = %ld\n",
                omp_get_thread_num(), numero - 1, subrespuesta);
            #pragma omp atomic
            respuesta += subrespuesta;
        }
    }
    tiempo_final = omp_get_wtime();  // Parar cronometro
    printf("El nomero %ld de la sucesion de Fibonacci es %ld\n", numero,
        respuesta);
    printf("Tiempo de ejecucion: %f segundos\n", tiempo_final -
       tiempo_inicio);
    return 0;
}
```

- Fibonacci con Tasks modificado:

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

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) {
    if (argc != 2) {
        printf("Uso: %s <numero_N_de_Fibonacci>\n", argv[0]);
        return 1;
    }
    int n, result;
    double tiempo_inicio, tiempo_final;
    n = atoi(argv[1]);

    tiempo_inicio = omp_get_wtime();  // Iniciar cronometro

    #pragma omp parallel
    {
        #pragma omp single
        result = fib(n);
    }
    tiempo_final = omp_get_wtime();  // Parar cronometro

    printf("El resultado de Fibonacci(%d) es %d\n", n, result);
    printf("Tiempo de ejecucion: %f segundos\n", tiempo_final -
       tiempo_inicio);

    return 0;
}
```

- Ejecución y Resultados:

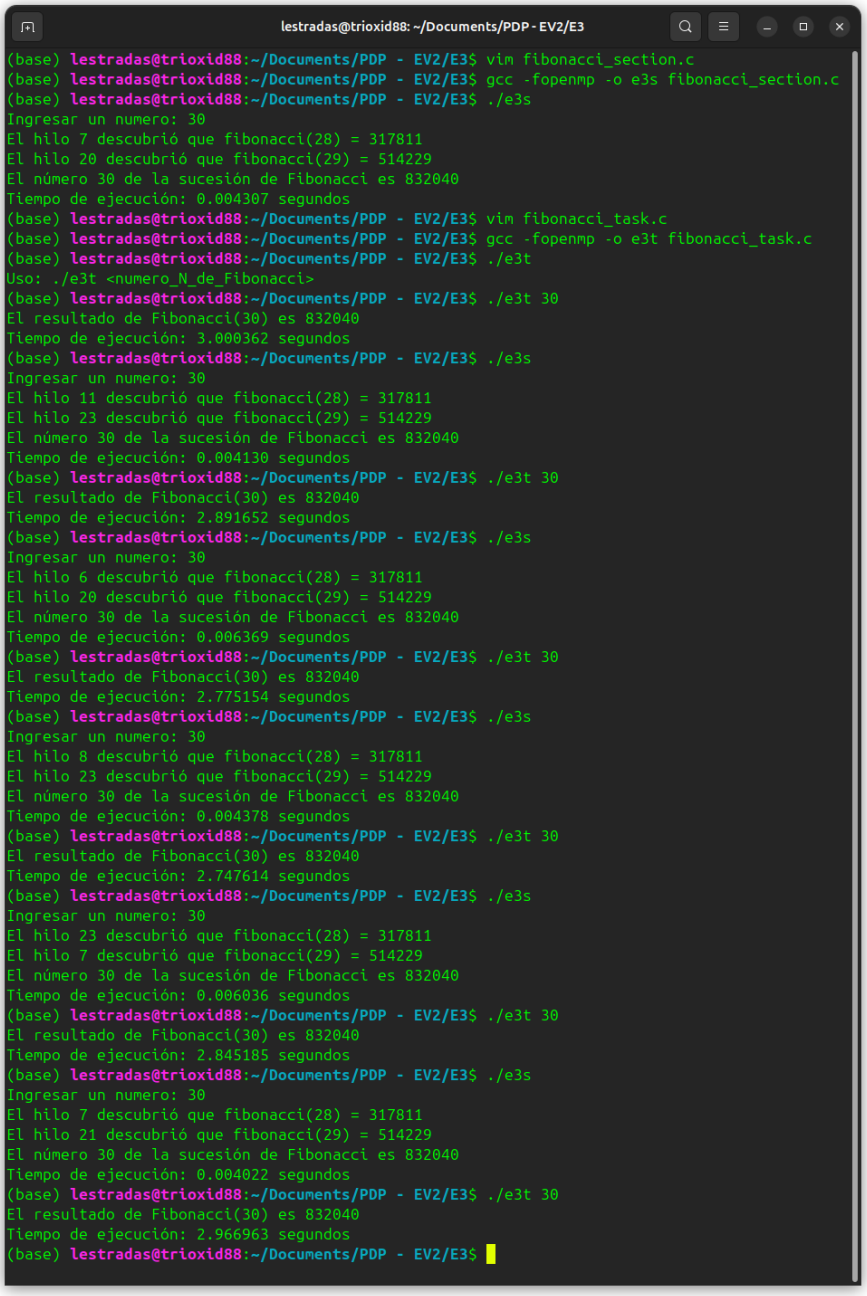

Figura 3: Ejecución y Resultados

- Análisis de los resultados obtenidos en la ejecución de los algoritmos de Fibonacci con sections y tasks para n=30.

  1. Observaciones sobre el tiempo de ejecución:

     - Algoritmo con sections:
       - Los tiempos de ejecución registrados oscilan entre 0.004037 segundos y 0.006369 segundos.
       - Estos tiempos son considerablemente bajos debido a la paralelización básica de la ejecución del cálculo de Fibonacci, que solo utiliza dos secciones (fibonacci(n-1) y fibonacci(n-2)), distribuyendo el trabajo entre dos hilos.
     - Algoritmo con tasks:
       - Los tiempos de ejecución registrados para este algoritmo están entre 2.747641 segundos y 3.000362 segundos.
       - A pesar de ser paralelizado con tareas, este enfoque introduce una mayor sobrecarga, ya que implica la creación de múltiples tareas dinámicas. Cada recursión crea nuevas tareas para fibonacci(n-1) y fibonacci(n-2), lo cual agrega complejidad en la gestión de las tareas.

  2. Desempeño comparativo entre los algoritmos:

     - Algoritmo con sections:
       - Este enfoque es extremadamente rápido para n=30 debido a su simplicidad. Solo divide el cálculo en dos partes (fibonacci(n-1) y fibonacci(n-2)) y deja que los hilos calculen esas partes en paralelo.
       - La eficiencia se debe a la limitación en la creación de solo dos secciones. Sin embargo, a medida que el valor de n aumenta, el tiempo de ejecución también aumentará significativamente, ya que el cálculo de Fibonacci es recursivo y no optimiza con memoización o estrategias similares.
     - Algoritmo con tasks:
       - Aunque la paralelización con tareas puede ser más escalable, para n=30, la sobrecarga de gestionar muchas tareas dinámicas es significativa.
       - A medida que aumenta el valor de n, este algoritmo tiene potencial para mejorar el desempeño, ya que puede distribuir las tareas más finamente entre los hilos, pero en este caso particular, la sobrecarga adicional de crear y sincronizar las tareas tiene un impacto negativo en el rendimiento.

3. Diferencias de desempeño y causa de las mismas:
    - Sobrecarga de gestión de tareas (tasks):
        - El enfoque basado en tasks genera muchas tareas dinámicas, lo cual introduce una sobrecarga considerable. Aunque esto puede ser beneficioso para valores más altos de n, en este caso (con n=30), la sobrecarga de tareas y la coordinación entre los hilos reduce la eficiencia.
        - Cada llamada recursiva en el algoritmo de tasks crea nuevas tareas para fibonacci(n-1) y fibonacci(n-2), lo que resulta en una gran cantidad de tareas a gestionar. El sistema de ejecución tiene que crear, ejecutar, y esperar a la finalización de las tareas (taskwait), lo que puede ralentizar el cálculo.
    - Simetría en las secciones (sections):
        - El enfoque de sections es mucho más simple y eficiente para valores moderados de n como n=30. Solo divide el problema en dos secciones, cada una de las cuales calcula fibonacci(n-1) y fibonacci(n-2), y esto se ejecuta en paralelo.
        - Este enfoque es muy rápido para valores pequeños o moderados de n, ya que no introduce mucha sobrecarga en la coordinación de hilos y secciones.
4. Número máximo donde los algoritmos presentan un desempeño similar:
    - Para valores de n alrededor de 30, el algoritmo de sections es claramente más rápido que el de tasks. Sin embargo, a medida que n crece, la eficiencia de sections disminuirá, mientras que tasks podría comenzar a beneficiarse de su escalabilidad.
    - Basado en el rendimiento observado, el número máximo donde ambos algoritmos pueden tener un desempeño similar podría estar alrededor de 20-25. A partir de n=30, como lo hemos visto, tasks introduce más sobrecarga y se vuelve más lento que sections.

5. Conclusión:
    - Mejor algoritmo para valores pequeños a moderados de n:
        - El algoritmo con sections es significativamente más rápido en este caso, ya que el cálculo de Fibonacci solo se divide en dos partes grandes, lo que minimiza la sobrecarga de paralelización.
    - Mejor algoritmo para valores grandes de n:
        - El algoritmo con tasks es más escalable y podría superar a sections para valores mayores de n. Sin embargo, para n=30, la sobrecarga es considerable y hace que sea más lento que sections.
    - Diferencia de desempeño:
        - La diferencia de desempeño entre ambos algoritmos se debe principalmente a la sobrecarga adicional que el manejo dinámico de tareas introduce en el enfoque con tasks. Para valores más pequeños de n, esta sobrecarga no se justifica, pero a medida que n aumenta, las tasks pueden distribuir mejor el trabajo y escalar mejor que sections.