Introducción a la Programación Paralela: Actividad OpenMP

Bernardo Zapico (62318)

bzapico@itba.edu.ar

Índice

Índice **1**
Introducción **2**
Serie de Fibonacci **2**
Explicación del Código y Directivas 2
Comparación Secuencial y Paralelo 5
Optimizaciones 6
Problema de las reinas **9**
Explicación del código 9
Paralelización 10
Varios Threads 11
Conclusión **13**
Anexo **14**
Fibonacci 14
Código secuencial 14
Código Paralelo de la cátedra 14
Iterativo 15
Memoization 16
Thresholding 16
Queens Puzzle 17
Secuencial 17
Paralelo 18
Optimización 19
Varios Threads 20

Introducción

En este proyecto se tomaron dos problemas y se buscó paralelizarlos mediante el uso de la librería OpenMP y los cores del sistema donde se realizaron las pruebas, en mi caso contaba con ocho cores. El objetivo del primer problema es, dado un número N, calcular el N-ésimo número de la serie de fibonacci y el objetivo del segundo problema es, dado un número N, calcular el número de soluciones para el problema de las reinas en un tablero NxN para N reinas. El código fuente donde se implementó las soluciones se encuentra en https://github.com/berni-245/IPP-TP1.

Serie de Fibonacci

Explicación del Código y Directivas

Para este problema se cuenta con el código ya paralelizado inicialmente por la cátedra en el enunciado, vamos a desglosarlo.

```
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);
}
```

Código 1: Función main del código de la cátedra.

Partamos primero por el main en el (Cod. 1), primero se recibe por argumento el valor de n y se lo transforma a un entero. Luego se corre la directiva #pragma omp parallel para un bloque de código, esta directiva lo que hace por defecto es crear un número fijo de Threads en un team para que corran el bloque de código que se especifique a continuación, empíricamente se verificó utilizando la función omp_get_num_threads() y 100 corridas que este número es igual al número de cores del sistema (al menos con configuración por defecto), en mi caso es ocho.
Luego en el bloque del parallel se corre la directiva #pragma omp single para una única línea de código que en esta se llama a la función fib(n) y guarda el resultado en una variable result. Esta directiva limita la línea a continuación (o bloque si hubiese) para que esta sea corrida por un único thread y adicionalmente agrega una barrera implícita

después de esa línea para que todos los demás threads no puedan avanzar con el código hasta que ese único thread haya terminado con esa línea, dejándolos en espera.

```
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;
  }
}
```

Código 2: Función fib(n) del código de la cátedra.

Ahora centrémonos en qué hace la función fib(n) (Cod. 2) y cómo saca provecho de los threads en espera previos. Si pensamos a la función de manera secuencial sin agregar paralelismo aún, para conseguir el valor en la posición N de la serie de fibonacci necesito calcular los valores en las posiciones N - 1 y N - 2 y sumarlos. Pero para calcular estos dos valores también necesitaré calcular sus 2 anteriores respectivos y así sucesivamente hasta llegar a los valores en los N < 2, cuyos valores son iguales a su N. En el código se representa esa idea, si N < 2 entonces el valor en la posición N en la serie es exactamente N, si N >= 2 entonces calculo los dos anteriores recursivamente y los sumo. Se puede ilustrar la idea con un árbol (Fig. 1) y tomando N=5 para simplificar.

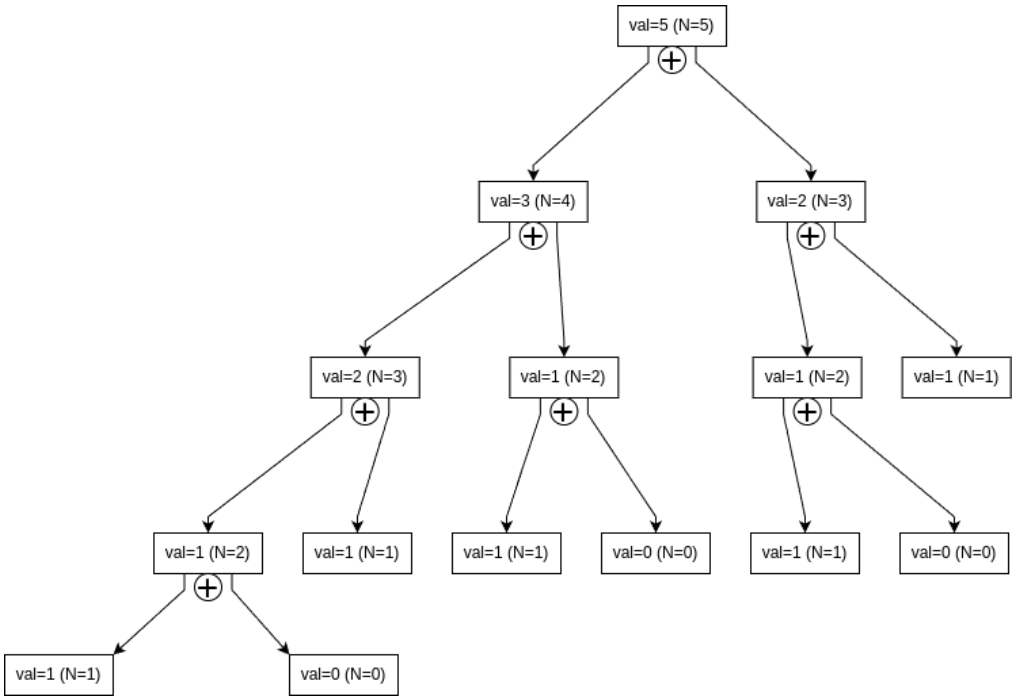

Figura 1: Árbol representando un cálculo recursivo de la función de fibonacci para N=5.

Notar como las hojas son los N=0 y N=1, que coincide con su valor, y como para N>=2 su valor coincide con la suma de los valores de sus dos nodos hijos.

Ahora volvamos al paralelismo. Inicialmente, a esta función entrará el thread que pasó la directiva con single anterior, este thread declarará unas variables i y j que se utilizarán para guardar los valores en la posición N-1 y N-2 de la serie respectivamente, pero para aprovechar el uso del paralelismo y delegar tareas a otros threads se creará una tarea por cada uno de esos dos cálculos parciales. Las tareas declaradas con #pragma omp task shared(var) podrán ser tomadas por Threads que estén en espera, por lo tanto, esos threads que dejamos esperando en el main podrán tomar estas tareas para optimizar los cálculos y paralelizar el esfuerzo. Es importante el uso del shared(var) en esas directivas para que el resultado guardado en las variables i y j pueda ser compartido con el thread que creó esas subtareas, caso contrario las modificaciones para dichas variables serán locales a los threads que tomaron cada subtarea y nunca se compartirán los cálculos con el thread creador. Finalmente se le dice al Thread creador con #pragma omp taskwait que espere a que terminen las 2 tareas creadas antes de continuar (puede tomar otras tareas mientras, esto evita deadlocks), esto se usa para esperar que las tareas guarden sus respuestas en las variables i y j antes de hacer su suma, y así evitar sumar valores incoherentes antes del guardado de los resultados parciales en esas variables.

Utilizando la función omp_get_thread_num() podemos obtener el id del thread actual, agregando un print al principio de la función fib podemos observar qué thread tiene cada tarea y tomando N=5, podemos armar un árbol visual de ejemplo (Fig. 2).

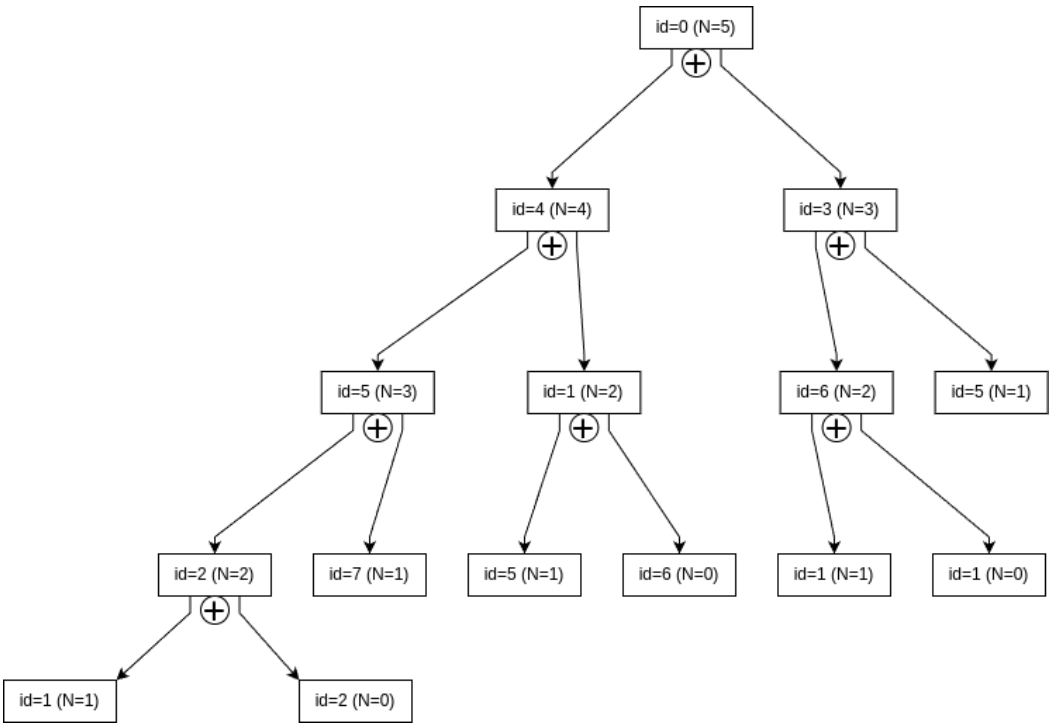

Figura 2: Árbol representando la distribución de Threads para el cálculo de la función de fibonacci.

Notar como distintos threads del team trabajan en distintos N y no solo el thread inicial (id=0) que fue el que entró en el bloque single. También notar como hijos de ciertos nodos fueron manejados por el mismo thread que creó la tarea, esto es porque cuando estos threads creadores se quedan en wait, pueden agarrar cualquier tarea de la cola de tareas pendientes del team.
En conclusión, se puede observar que la intención del código es subdividir cálculos lo máximo posible (hasta llegar a las hojas de árbol) para luego fusionar los resultados y formar el resultado completo.

Comparación Secuencial y Paralelo

Para la ejecución del código Paralelo se necesita incluir el argumento `-fopenmp` para que la librería de OpenMP detecte las directivas en el código, en cambio para ejecutar el código secuencial basta con no incluir dicho argumento y las directivas serán ignoradas. En la práctica como se utilizó la función `omp_get_wtime()` para medir el tiempo, el flag tuvo que ser incluido siempre. Las pruebas fueron realizadas en el siguiente Jupyer Notebook dentro del repositorio mencionado previamente.

Inicialmente no parecía muy notable que había una diferencia entre el código Secuencial y Paralelo para N chicos, por lo que se decidió hacer una gráfica lado a lado entre el tiempo de ejecución medio secuencial y paralelo para todos los N entre 0 y 40. Para calcular dicha media se registró el tiempo de ejecución de 5 pruebas para cada N y se calculó su promedio y su desvío estándar. Finalmente se graficaron los tiempos en escala logarítmica en dos errorbar (Fig. 3).

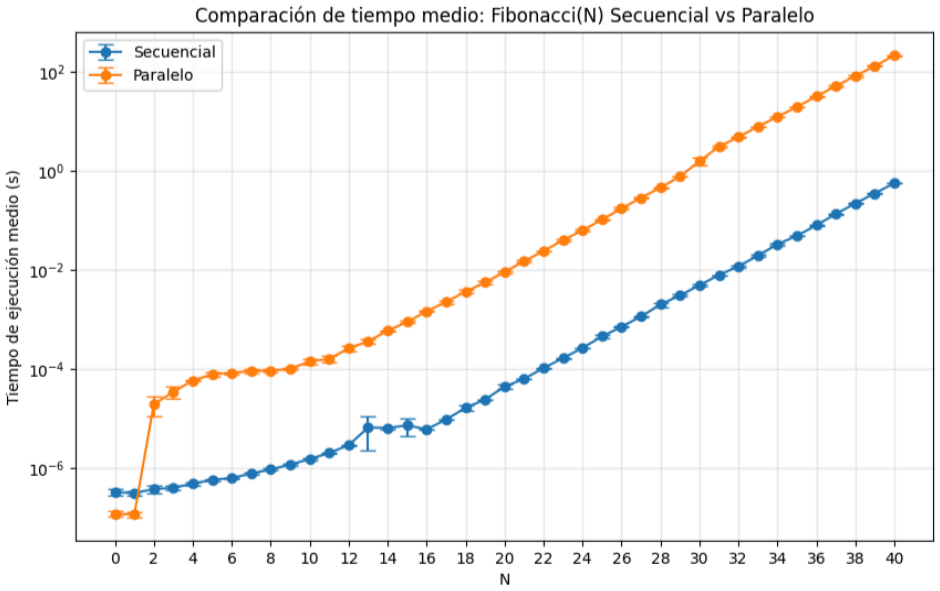

Figura 3: Comparación entre tiempos de ejecución medio entre el algoritmo Fib(n) secuencial y paralelo.

Se observa como el tiempo de ejecución del algoritmo secuencial es siempre más rápido que el paralelo para N>=2, a pesar de que este último está trabajando en varios threads. Esto se debe a varios factores:

- Toda llamada recursiva a fib(n) se convierte en una tarea independiente. Esto escala muy mal cuando N es grande ya que se generan millones de tareas que deben distribuirse en un número limitado de Threads (ocho en este caso).
- Por el mismo motivo, se generan muchos costos de cambio de tarea y costos de creación que superan el costo del trabajo secuencial.
- Por el mismo motivo, la directiva #pragma omp taskwait genera muchos costos de sincronización y espera de resultados pues se realiza en todos los niveles del árbol recursivo, tanto para N grandes como N chicos.
- El costo de llamados recursivos afecta a los dos, se requiere una constante reserva de espacio de pila para cada llamado a función.

Optimizaciones

Para lograr mejores tiempos, administración de recursos y aprovechar el uso de varios threads, se implementaron tres optimizaciones clásicas distintas para los problemas recursivos.

La primera optimización fue transformar el código en **iterativo**, básicamente se cambió la función fib(n) para que el cálculo cuando N es mayor o igual a 2 sea haga en un for donde se parte sabiendo los valores de la serie para N=0 y N=1 y se va calculando parcialmente el valor de la serie de fibonacci para cada N una única vez hasta llegar al N pedido. Esto sigue un principio de programación dinámica ya que se evita recálculos de los N-1 y N-2 para cada N y por lo tanto mejora bastante los tiempos de ejecución. El código en cuestión se encuentra adjuntado en el anexo junto a todos los otros.

Se corrió 5 pruebas para cada N distintos para sacar una media apropiada y se puso a prueba cómo rinden los algoritmos secuencial, paralelo (de la cátedra) y iterativo (Fig. 4).

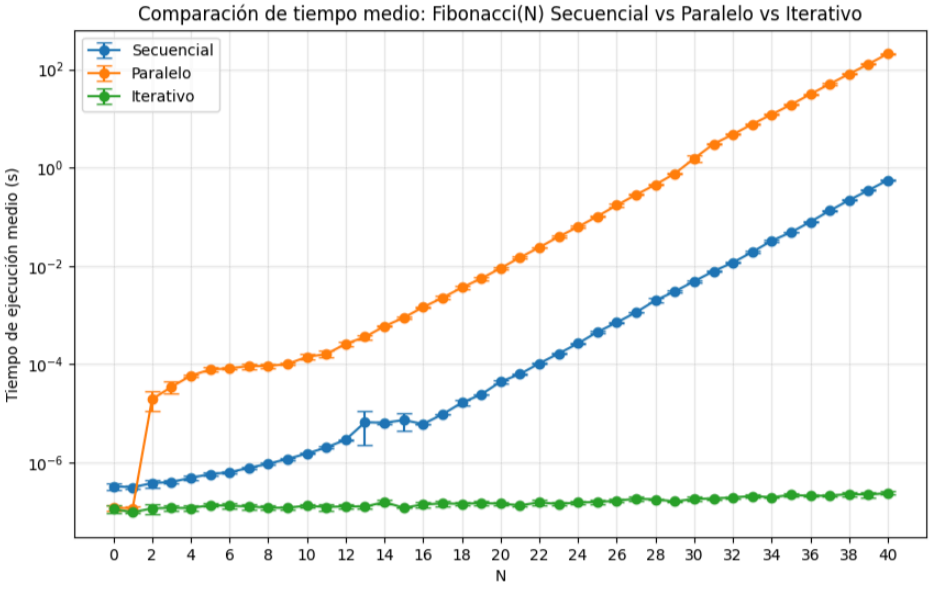

Figura 4: Tiempos de ejecución medios de Fib(N) en comparación con el algoritmo iterativo para cada N entre 0 y 40.

En los resultados obtenidos se puede notar que el algoritmo iterativo mejora considerablemente el tiempo de ejecución, a pesar de que este no sea paralelizable pues depende de los valores anteriores para calcular cada N.

La segunda optimización realizada fue agregarle un mecanismo de **Thresholding**, es decir, que a partir de cierto N < Threshold el código pase a ser secuencial. En mi caso se utilizó una fórmula para calcular el Threshold:

$$threshold = max(20, floor(\frac{3}{4}N))$$

Se definió dicho threshold de manera dinámica en base al N con el propósito de que el cálculo del threshold dependa de qué tan grande es el N. Luego se puso a prueba los tiempos de ejecución en para cada N en comparación con el secuencial y el paralelo (Fig. 5).

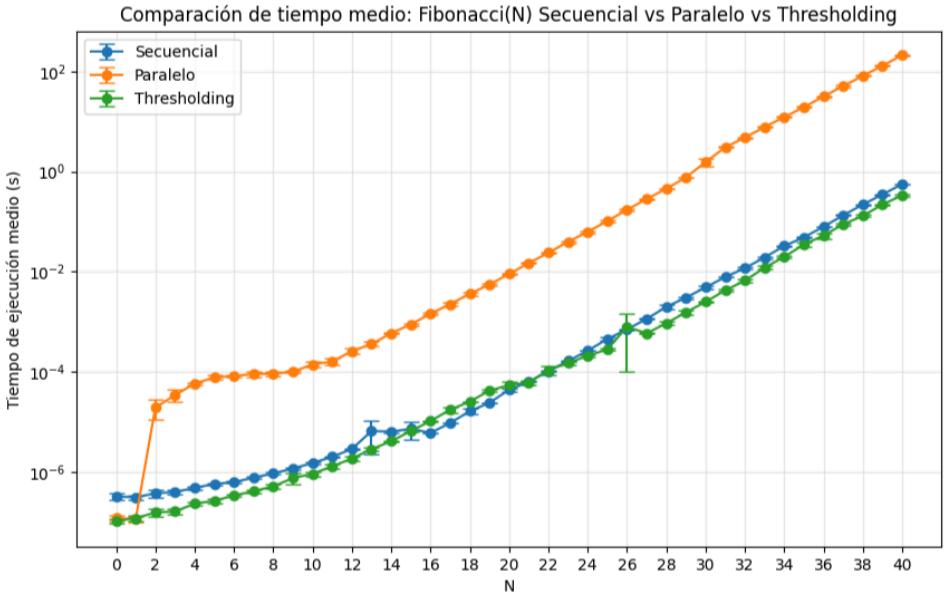

Figura 5: Tiempos de ejecución medios de Fib(N) en comparación con el algoritmo de threshold para cada N entre 0 y 40.

Se observan mejoras sobre el algoritmo paralelo original pero aún así sus tiempos de ejecución siguen siendo bastante parecidos al secuencial.

La tercera optimización realizada fue agregar un sistema de **memoization**, es decir, que se guarde los resultados parciales en un arreglo para evitar recalcularlos todo el tiempo. Pensemos en la estructura del árbol del algoritmo recursivo para N=5 (Fig. 1) y cómo tiene que calcular el valor de `fib(N=3)` que aparece dos veces, lo que hace es calcularlos usando sus dos subárboles de llamados recursivos para N=2 y N=1, esto lo va a hacer tantas veces como se necesite calcular el `fib(3)` (que en un caso de `fib(N)` con N más grande podrían ser muchas más veces), en cambio con el sistema de memoization este cálculo se hará una única vez y en futuros llamados a `fib(3)` se accede al valor calculado la primera vez, esto también sigue un principio de programación dinámica. Luego se puso a prueba los tiempos de ejecución de este

algoritmo con los algoritmos paralelos y secuencial (Fig. 6). Donde se observa una considerable mejora en el tiempo tardado para conseguir los resultados.

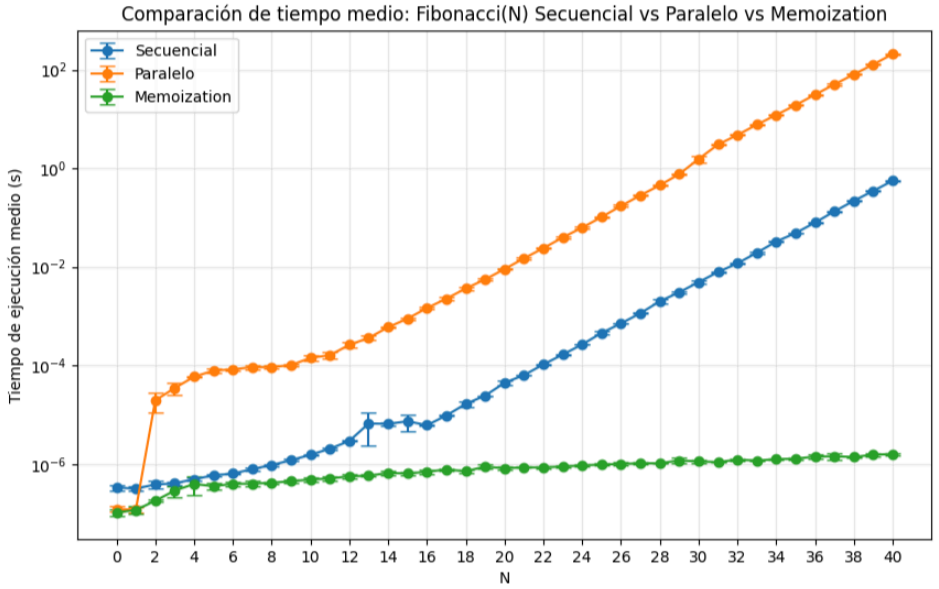

Figura 6: Tiempos de ejecución medios de Fib(N) en comparación con el algoritmo de memoization para cada N entre 0 y 40.

Finalmente se puso en comparación a todos los algoritmos juntos (Fig. 7), donde se puede concluir que el algoritmo iterativo fue la mejor optimización para este problema, mientras que el algoritmo de paralelización original fue el peor.

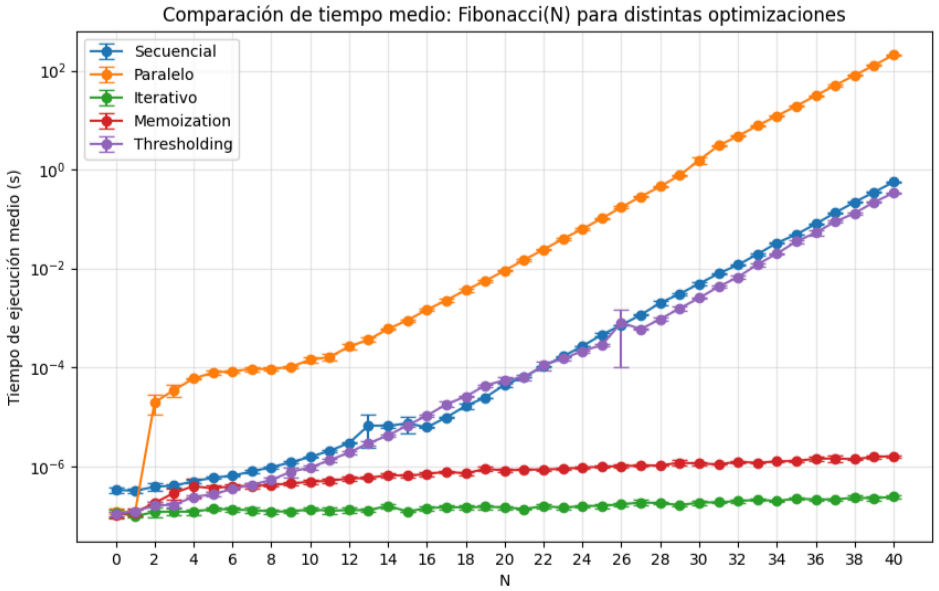

Figura 7: Tiempos de ejecución medios de Fib(N) para todos los algoritmos para cada N entre 0 y 40.

Problema de las reinas

Explicación del código

En el problema de las N-reinas se tienen un tablero NxN y se busca distribuir las N reinas en dicho tablero sin que ninguna se esté amenazando entre sí. El código busca contar la cantidad de soluciones para un tablero NxN.
Tomando un N=8 arbitrario, imaginemos un tablero de ajedrez 8x8.

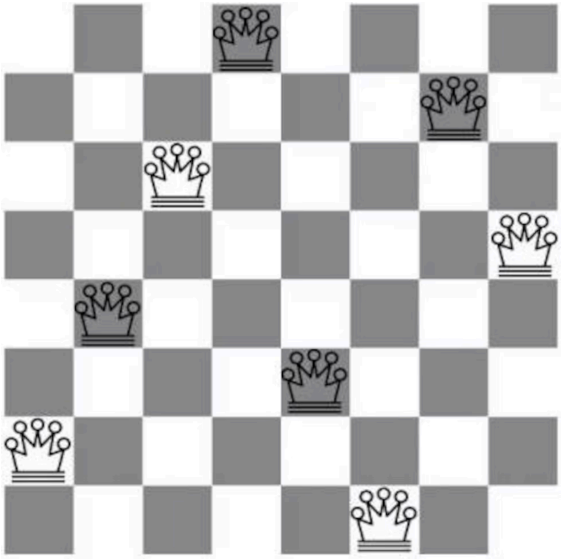

Figura 8: Representación visual del tablero para un problema de las reinas con N=8.

La idea del código secuencial es:

- Pararse en la primera fila.
- Ir por cada columna en la primera fila y busco el primer espacio `(row, col)` que pueda colocar una reina sin que se amenace con otra reina, la coloco (se guarda en un arreglo) y paso a la siguiente fila.
- En la siguiente fila voy columna por columna hasta encontrar el primer espacio que pueda color una reina sin conflictos.
- Y así sucesivamente hasta que pueda colocar una reina en la última fila y, de poder, sumo un contador con la cantidad de soluciones.
- En caso de haber completado el tablero o de no poder poner ninguna ficha en toda una fila, vuelvo al stack de la fila anterior y sigo avanzado por las columnas desde donde me quedé la última vez en esa fila anterior y sigo, se va a volver en la pila tanto como sea necesario para poder continuar.
- Finalmente, cuando recorrí todas las posiciones, termina el algoritmo y devuelvo la cantidad de soluciones.

Paralelización

Para la paralelización del código, se buscó crear un sistema de tareas similar al código de la cátedra para fibonacci. Lo que se hizo fue crear un team de threads al iniciar la aplicación, que un solo thread pase a llamar a la función `solve(row=0, *board)` y que este empiece a lanzar tareas para que cualquier thread en espera pueda tomarla. El punto clave es dónde se lanza la tarea y para el código inicial estas son lanzadas para cada par `(row, col)` válido, es decir, si pasan el control de la función `is_safe` entonces se crea una tarea para que controle el flujo de la siguiente fila tras poner la reina en la fila actual. Similar al caso de fibonacci, se probó para distintos N en ambos algoritmos y se ejecutó 3 veces para cada N para lograr una media y desvío de los datos (Fig. 9).

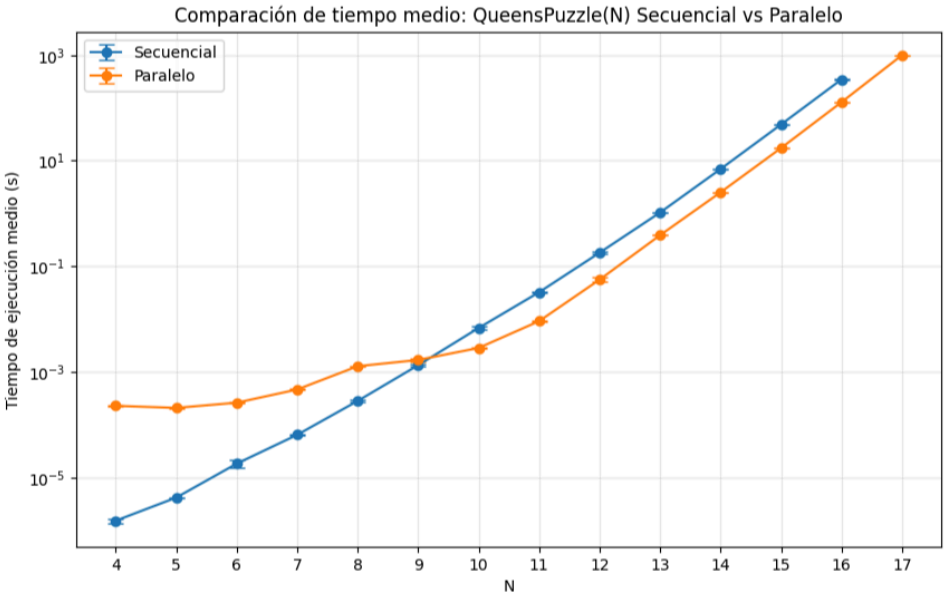

Figura 9: Tiempos de ejecución medios de queensProblem(N) para algoritmos secuencial y paralelo para cada N entre 4 y 17.

Se puede notar que los tiempos de ejecución para los últimos N crece significativamente, también se observa que el algoritmo paralelo propuesta es mejor que el secuencial para N>9.

Se intentó optimizar el algoritmo paralelo mencionado limitando la cantidad de tareas lanzadas. Esto se hizo separando en una función aparte el recorrido de la primera fila y en lugar de lanzar tareas por cada par `(row, col)` válido, se lanzó para todas las columnas de la primera fila únicamente (todas estas son pares `(row=0, col)` válidos) y que ahí siga secuencial, este algoritmo lo llamo `First Row`. En esta función separada se utilizó la instrucción `#pragma omp parallel for reduction(+:solutions)` que lanza en una cola de tareas cada iteración del for, también se agregó la instrucción `reduction(+:solutions)` al for para indicarle que en cada iteración se va a sumar una variable `solutions`, que es el contador de

soluciones. Después se analizó el tiempo de ejecución medio tardado en comparación con el método de paralelización original (Fig. 10).

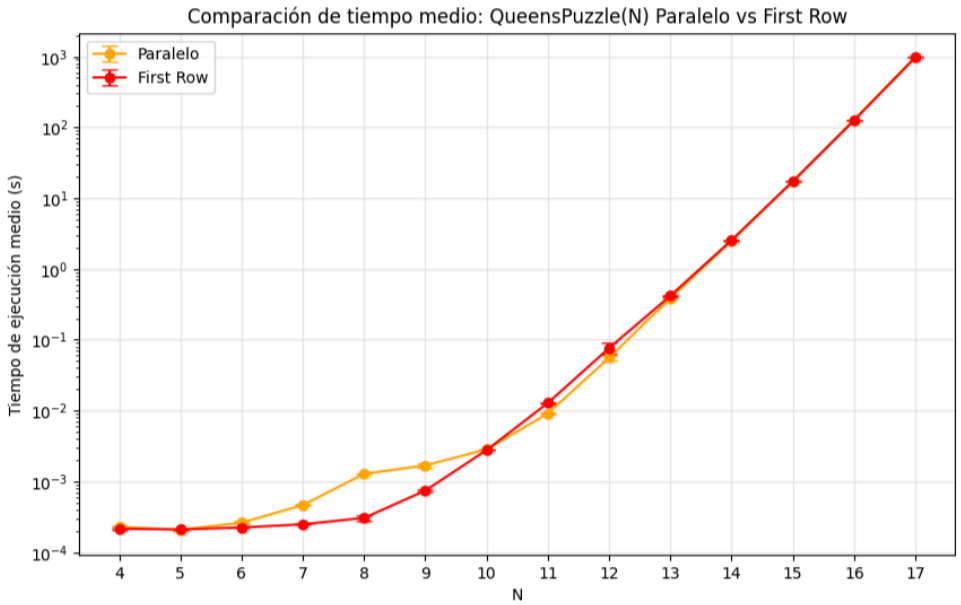

Figura 10: Tiempos de ejecución medios de queensProblem(N) para algoritmos paralelo original y paralelo sólo en la primera fila para cada N entre 4 y 17.

Como se puede observar, no hubo mucha diferencia entre las soluciones paralelas propuestas, esto se puede deber a que no se probó con N muy grandes ya que si vemos el gráfico se observa una ligera tendencia de que el algoritmo `First Row` empieza a ser ligeramente más rápido que el paralelo original a mayor N.

Varios Threads

Finalmente para el problema de las N reinas se decidió ver cómo afecta la cantidad de hilos al rendimiento del algoritmo, por lo que se decidió modificar la cantidad de hilos creados con la instrucción `parallel` para que varíe entre 1 y el máximo número de cores en el sistema, en mi caso 8. Esto se puede hacer con la instrucción `omp_set_num_threads(threadAmount)`. Luego se comparó los tiempos de ejecución medios para cada número de threads del programa con N entre 4 y 15 y se lo graficó (Fig. 11).

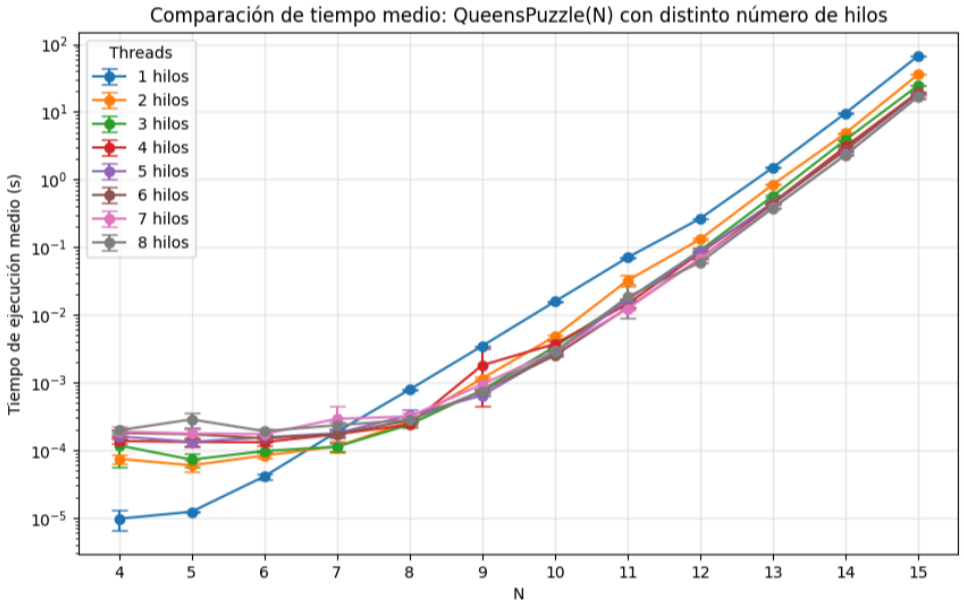

Figura 11: Tiempos de ejecución medios de queensProblem(N) con el algoritmo First Row para distintos Threads con N entre 4 y 17.

Como se puede observar, a mayor número de hilos haya que se acerque al valor del número de cores del sistema, se consigue un mejor tiempo de ejecución.

Conclusión

El trabajo permitió contrastar dos problemas de naturaleza distinta y analizar cómo responden frente a la paralelización con OpenMP. En el primero, la paralelización se aplicaba sobre subcálculos dependientes entre sí, lo que generaba gran cantidad de tareas pequeñas con fuerte necesidad de sincronización. En el segundo, la paralelización consistía en dividir tareas más independientes y de mayor costo computacional, cuyo resultado final se integraba únicamente en un contador compartido.

En el problema de Fibonacci, se observó que el paralelismo recursivo mal planteado genera más costos que beneficios: la explosión combinatoria de llamadas, la creación masiva de tareas y las sincronizaciones constantes hacen que la versión paralela sea más lenta que la secuencial. Esto pone en evidencia que no siempre el paralelismo es la mejor estrategia y que, en ciertos casos, optimizaciones algorítmicas como la programación dinámica o la memoización son mucho más efectivas.

En el problema de las N-reinas, en cambio, la paralelización sí mostró mejoras. Al distribuir el conteo de soluciones entre varios hilos, el tiempo de ejecución disminuyó frente al secuencial, especialmente para valores de N menores a 18. Además, se comprobó que el rendimiento escala con la cantidad de hilos hasta el límite impuesto por los cores del sistema.

En resumen, estos resultados muestran la importancia de elegir cuidadosamente qué problemas paralelizar, mientras algunos algoritmos se benefician enormemente del trabajo concurrente, en otros el costo de administrar las tareas supera las ventajas. El paralelismo no reemplaza a un buen diseño algorítmico, sino que debe considerarse como una herramienta complementaria que rinde mejor en problemas con alta independencia entre subtareas.

Anexo

En este anexo se incluyen los códigos fuentes con los que se hicieron las pruebas.

Fibonacci

Código secuencial

```c
// Only reason I created this file instead of removing the -fopenmp flag is because I used the omp_get_wtime() function for the time
// and if I remove the flag, then it will fail at compile time due to that function not being declared

#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;
  i=fib(n-1);
  j=fib(n-2);

  return i+j;
}

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

  double start = omp_get_wtime();
  result = fib(n);
  double end = omp_get_wtime();
  printf("%.15f\n", end - start);
  // printf("Result is %d\n", result);

  return 0;
}
```

Código 3: Código secuencial de Fib(N).

Código Paralelo de la cátedra

```c
#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
      {
         double start = omp_get_wtime();
         result = fib(n);
         double end = omp_get_wtime();
         printf("%.15f\n", end - start);
      }
   }
   // printf("Result is %d\n", result);

   return 0;
}
```

Código 4: Código paralelo de la cátedra de Fib(N).

Iterativo

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

int fib(int n) {
   if (n < 2) return n;

   int previousToLast = 0, last = 1, aux = 0;

   for (int i = 2; i <= n; i++) {
      aux = previousToLast + last;
      previousToLast = last;
      last = aux;
   }
   return aux;
}

int main(int argc, char **argv) {
   int n, result;
   n = atoi(argv[1]);
   double start = omp_get_wtime();
   result = fib(n);
   double end = omp_get_wtime();
   printf("%.15f\n", end - start);
   // printf("Result is %d\n", result);

   return 0;
}
```

Código 5: Código Iterativo de Fib(N).

Memoization

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

#define MAX 50
int values[MAX] = {0};
omp_lock_t locks[MAX];

int fib(int n) {
  if (n < 2)
    return n;
  omp_set_lock(&locks[n]); // ensure only one thread is calculating the val of fib(n) at the time

  if (values[n] != 0) { // already calculated
    int val = values[n];
    omp_unset_lock(&locks[n]);
    return val;
  }

  int i = fib(n - 1);
  int j = fib(n - 2);
  int result = i + j;

  values[n] = result;

  omp_unset_lock(&locks[n]);
  return result;
}

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

  for (int k = 0; k < MAX; k++)
    omp_init_lock(&locks[k]);

#pragma omp parallel
  {
#pragma omp single
    {
      double start = omp_get_wtime();
      result = fib(n);
      double end = omp_get_wtime();
      printf("%.15f\n", end - start);
    }
  }
  // printf("Result is %d\n", result);

  for (int k = 0; k < MAX; k++)
    omp_destroy_lock(&locks[k]);

  return 0;
}
```

Código 6: Código paralelo con Memoization de Fib(N).

Thresholding

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

#define MAX(a, b) (((a) > (b)) ? (a) : (b))

int fib(int n, const int nThreshold) {
  int i, j;
  if (n<2)
    return n;
  if (n < nThreshold)
    return fib(n-1, nThreshold) + fib(n-2, nThreshold);
#pragma omp task shared(i)
    i=fib(n-1, nThreshold);
#pragma omp task shared(j)
    j=fib(n-2, nThreshold);
#pragma omp taskwait
    return i+j;
}

int main(int argc, char **argv){
  int n, nThreadhold, result;
  char *a = argv[1];
  n = atoi(a);
  nThreadhold = MAX(20, (int) (n*0.75));

#pragma omp parallel
  {
#pragma omp single
    {
      double start = omp_get_wtime();
      result = fib(n, nThreadhold);
      double end = omp_get_wtime();
      printf("%.15f\n", end - start);
    }
  }
  // printf("Result is %d\n", result);

  return 0;
}
```

Código 7: Código Paralelo con Thresholding de Fib(N).

Queens Puzzle

Secuencial

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

long long solutions = 0;
int N;

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 argc, char **argv) {
  char *a = argv[1];
  N = atoi(a);
  int board[N];
  double start = omp_get_wtime();
  solve(0, board);
  double end = omp_get_wtime();
  // printf("Soluciones para %d reinas: %lld\n", N, solutions);
  printf("%.15f\n", end - start);
  return 0;
}
```

Código 8: Código secuencial de queensPuzzle(N) de la cátedra.

Paralelo

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

long long solutions = 0;
int N;

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) {
#pragma omp atomic
    solutions++;
    return;
  }

  for (int col = 0; col < N; col++) {
    if (is_safe(board, row, col)) {
      int *new_board = malloc(N * sizeof(int));
```

```
        memcpy(new_board, board, row * sizeof(int));
        new_board[row] = col;

#pragma omp task firstprivate(row, new_board)
        {
          solve(row + 1, new_board);
          free(new_board);
        }
      }
    }
}

int main(int argc, char **argv) {
  char *a = argv[1];
  N = atoi(a);
  int board[N];
  double start = omp_get_wtime();
#pragma omp parallel
  {
#pragma omp single
    {
      solve(0, board);
    }
  }

  double end = omp_get_wtime();
  // printf("Soluciones para %d reinas: %lld\n", N, solutions);
  printf("%.15f\n", end - start);
  return 0;
}
```

Código 9: Código paralelo inicial de queensPuzzle(N).

Optimización

```
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <omp.h>
#define FIRST_COL 0

long long solutions = 0;
int N;

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) {
#pragma omp atomic
    solutions++;
    return;
  }

  for (int col = 0; col < N; col++) {
    if (is_safe(board, row, col)) {
```

```
        board[row] = col;
        solve(row + 1, board);
      }
    }
}

void n_queens() {
#pragma omp parallel for reduction(+:solutions)
  for (int col = FIRST_COL; col < N; col++) {
    int board[N];
    board[FIRST_COL] = col;
    solve(FIRST_COL + 1, board);
  }
}



int main(int argc, char **argv) {
  char *a = argv[1];
  N = atoi(a);
  int board[N];
  double start = omp_get_wtime();
  n_queens();
  double end = omp_get_wtime();
  // printf("Soluciones para %d reinas: %lld\n", N, solutions);
  printf("%.15f\n", end - start);
  return 0;
}
```

Código 10: Código paralelo en la primera fila de queensPuzzle(N).

Varios Threads

```
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <omp.h>
#define FIRST_COL 0

long long solutions = 0;
int N;

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) {
#pragma omp atomic
    solutions++;
    return;
  }

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

```
            solve(row + 1, board);
        }
    }
}

void n_queens() {
#pragma omp parallel for reduction(+:solutions)
    for (int col = FIRST_COL; col < N; col++) {
        int board[N];
        board[FIRST_COL] = col;
        solve(FIRST_COL + 1, board);
    }
}



int main(int argc, char **argv) {
    char *a = argv[1];
    N = atoi(a);
    char *b = argv[2];
    int threadAmount = atoi(b);
    omp_set_num_threads(threadAmount);
    double start = omp_get_wtime();
    n_queens();
    double end = omp_get_wtime();
    // printf("Soluciones para %d reinas: %lld\n", N, solutions);
    printf("%.15f\n", end - start);
    return 0;
}
```

Código 11: Código paralelo en la primera fila de queensPuzzle(N) para probar con distintos threads.