Project: Parallelizing Sorting Algorithms

Raúl Isaí Oriza Leal
Alejandra Ramón Mendoza, Miriam Berenice López Plaza

03 de abril del 2023

Abstracto

The use of parallel sorting algorithms in the C language is a widely used technique in industry and academia to improve the efficiency and performance of computer systems. Parallel sorting algorithms are based on dividing a set of data into smaller subsets that can be independently sorted in different threads of execution. Subsequently, these subsets are combined into one, creating a complete sorted list.

The most commonly used parallel sorting algorithm is the Quicksort algorithm. Although its time and space complexity is the same as the sequential version, its execution speed is increased thanks to the use of multiple processors. Another algorithm used in parallel is Mergesort, which is highly scalable and works well in distributed and parallel architectures.

Despite the advantages of using parallel sorting algorithms in the C language, it is important to consider some critical aspects, such as thread synchronization and shared memory management, to avoid errors and ensure proper operation. In general, the use of parallel sorting algorithms in the C language is an effective technique to improve the performance of computer systems in situations where large amounts of data are handled.

Índice de figuras

1. Algoritmo Quicksort 4
2. #pragma omp parallel en función a Quicksort() 4
3. Sección paralelo con asignación de hilos en función main() 5
4. Algoritmo Merge Sort 6
5. Análisis de resultados del programa QuickSort en programación paralelo 8
6. Quicksort serial con n=1,000 18
7. Quicksort serial con n=10,000 18
8. Quicksort serial con n=100,000 18
9. Quicksort serial con n=200,000 19
10. Quicksort serial con n=300,000 19
11. Quicksort paralelo con 2 hilos y 1000 elementos 19
12. Quicksort paralelo con 4 hilos y 1000 elementos 19
13. Quicksrt paralelo con 8 hilos y 1000 elementos 19
14. Merge sort paralelo con 2 hilos y 10 elementos 19
15. Merge sort paralelo con 2 hilos y 1000 elementos 20
16. Merge sort paralelo con 4 hilos y 1000 elementos 20
17. Merge sort paralelo con 8 hilos y 1000 elementos 20
18. Merge sort paralelo con 16 hilos y 1000 elementos 20
19. Diagrama de flujo Quicksort 21
20. Diagrama de flujo Mergesort main() 22
21. Diagrama de flujo Mergesort funciones 23

Índice de tablas

1. Serial Quick sort 8
2. Comparación tiempo Parallel Quick sort 9
3. Comparación tiempo Parallel Merge sort 10

Planteamiento del problema

El manejo de datos en un programa es de vital importancia para su funcionamiento, en especial cuando se tienen grandes cantidades. Estos datos son empleados por otros programas o incluso por el usuario y cuando se requiere hacer una cantidad considerable de búsquedas y es importante el factor tiempo. Ordenar un grupo de datos significa mover los datos o sus referencias para que queden en una secuencia por categorías y en forma ascendente o descendente.

Justificación

Tanto el enfoque de memoria distribuida como el de memoria compartida tienen sus ventajas y desventajas, y la elección del enfoque adecuado depende en gran medida de las necesidades específicas de la aplicación.

El enfoque de memoria distribuida se utiliza cuando se necesita una alta escalabilidad y se trabaja con grandes conjuntos de datos. En este, los datos se dividen en fragmentos y se distribuyen entre diferentes nodos en un sistema distribuido, lo que permite procesar grandes conjuntos de datos de manera eficiente y paralela. Además, el enfoque de memoria distribuida también proporciona una mayor tolerancia a fallos, ya que los datos están distribuidos y replicados en múltiples nodos, lo que reduce la probabilidad de una pérdida de datos debido a un fallo en un solo nodo.

Por otro lado, el enfoque de memoria compartida se utiliza cuando se necesita una mayor coherencia y consistencia de datos. En este, todos los nodos comparten el mismo espacio de memoria, lo que permite el acceso directo a los datos y una mayor eficiencia en la comunicación entre nodos. Además, el uso de memoria compartida también permite una mayor coherencia de datos, ya que todos los nodos pueden acceder y actualizar los mismos datos de manera consistente.

En resumen, el enfoque de memoria distribuida es adecuado para aplicaciones que requieren alta escalabilidad y tolerancia a fallos, mientras que el enfoque de memoria compartida es adecuado para aplicaciones que requieren una mayor coherencia y consistencia de datos. La elección adecuada del enfoque de memoria puede mejorar significativamente el rendimiento y la eficiencia de una aplicación.

Objetivo

Implementar algoritmos de ordenamiento empleando programación en paralelo, con el fin de ordenar una determinada cantidad de datos en el menor tiempo posible.

1. Descripción Detallada Del Procedimiento

El entorno empleado para este proyecto es Visual Studio Code, el cual es un editor de código que permite escribir, editar, compilar, depurar y ejecutar los programas que se analizan.

1.1. Quicksort

El algoritmo de Quicksort utiliza la estrategia de divide y conquista para ordenar un arreglo. Primero, se elige un elemento del arreglo como pivote (en este caso, el último elemento), y se divide el arreglo en dos partes: los elementos que son menores o iguales que el pivote y los elementos que son mayores que el pivote. Luego, se ordena recursivamente cada una de estas partes utilizando el mismo algoritmo. El proceso de división y ordenamiento recursivo continúa hasta que el arreglo esté completamente ordenado.

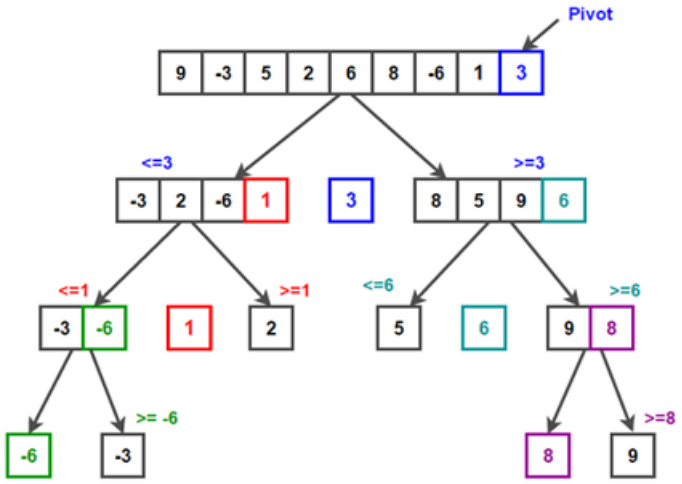

Figura 1: Algoritmo Quicksort

El arreglo de números a ordenar esta dado por una función que llena el arreglo con números aleatorio entre el 0 y 100 hasta n números que se asignan dentro del programa.
En esta implementación, utilizamos la directiva *#pragma omp parallel* para crear un equipo de hilos que ejecutan el algoritmo de Quicksort de forma paralela. Además, utilizamos la directiva *#pragma omp sections* para dividir la tarea de ordenamiento en dos secciones que pueden ejecutarse simultáneamente en diferentes hilos.

```
#pragma omp parallel sections
{
    #pragma omp section
    quicksort(arr, low, pivotIndex - 1);
    #pragma omp section
    quicksort(arr, pivotIndex + 1, high);
}
```

Figura 2: #pragma omp parallel en función a Quicksort()

La directiva *#pragma omp single* se utiliza para asegurar que solo un hilo ejecute la llamada a la función quicksort en el nivel superior de recursión. Esto se debe a que no tiene sentido ejecutar la misma tarea en paralelo en todos los hilos.

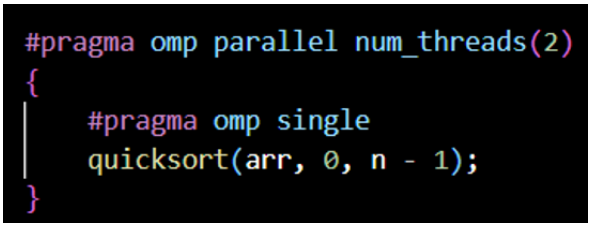

Figura 3: Sección paralelo con asignación de hilos en función main()

Finalmente, utilizamos la directiva $\#pragmaompparallelnum_threads()$ para especificar que queremos utilizar dos hilos para ejecutar el programa.

Es importante mencionar que la eficiencia de la programación en paralelo depende de varios factores, como el tamaño del arreglo a ordenar y la cantidad de núcleos de procesamiento disponibles. Es posible que en algunos casos, la versión en paralelo no sea más rápida que la versión secuencial debido a la sobrecarga introducida por la programación en paralelo.

1.2. Merge Sort

El Merge Sort es un algoritmo de ordenamiento eficiente que divide repetidamente la lista a ordenar en mitades iguales y luego combina las sub-listas ordenadas para producir una lista ordenada completa. El proceso del Merge Sort se puede describir en los siguientes pasos:

1. División: se divide la lista original en dos sub-listas de tamaño similar. Este proceso se realiza recursivamente hasta que cada sub-lista tiene un solo elemento.

2. Ordenamiento: se ordenan las sub-listas mediante el algoritmo Merge Sort, aplicando la misma estrategia de división y ordenamiento recursivo.

3. Mezcla: se mezclan las sub-listas ordenadas en una sola lista ordenada. Para hacer esto, se compara el primer elemento de cada sub-lista y se coloca el elemento menor en la lista de salida. Este proceso se repite hasta que todas las sub-listas se hayan mezclado en una sola lista ordenada.

4. La lista resultante es la lista original ordenada.

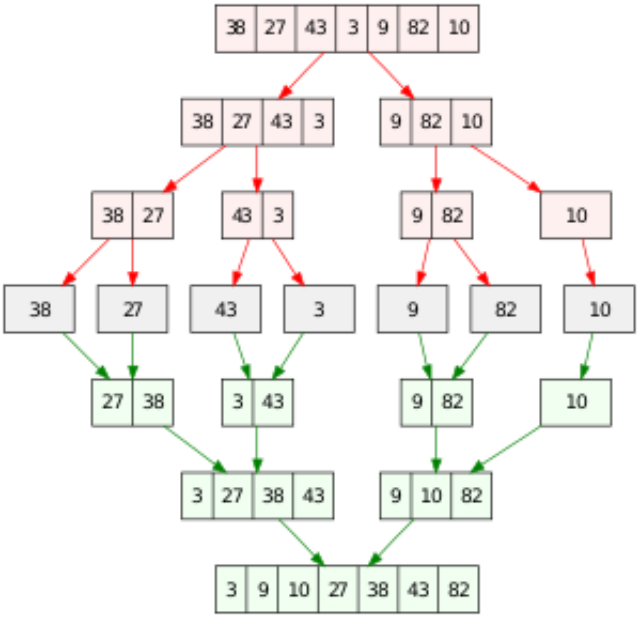

Figura 4: Algoritmo Merge Sort

Este proceso se repite recursivamente para cada sub-lista hasta que se llega a la lista completa ordenada.

2. Diseño Detallado De Los Módulos

2.1. Quick Sort

Los módulos del Quick sort son los siguientes:

- **Bibliotecas:** el código comienza importando las bibliotecas necesarias para su funcionamiento: stdio.h, stdlib.h, omp.h y time.h. La biblioteca stdio.h se utiliza para entrada/salida estándar, stdlib.h para funciones generales, omp.h para el uso de OpenMP y time.h para la medición del tiempo de ejecución.
- **Constantes:** se define la constante n que se utiliza como tamaño del arreglo de entrada.
- **Función quicksort:** esta función es la encargada de implementar el algoritmo de ordenación quicksort. Toma tres argumentos: un arreglo de enteros arr, y dos enteros low y high que indican los límites inferior y superior de la porción del arreglo que se desea ordenar. Dentro de la función se realiza la partición del arreglo y se ordenan recursivamente las dos sub-particiones resultantes. Se utiliza la directiva *#pragma omp parallel sections* para ejecutar cada llamada recursiva en una sección paralela.

- **Función partition:** esta función es llamada por la función quicksort y se encarga de realizar la partición del arreglo arr en dos sub-arreglos, uno con elementos menores o iguales al pivote y otro con elementos mayores al pivote. Toma tres argumentos: un arreglo de enteros arr y dos enteros low y high que indican los límites inferior y superior de la porción del arreglo que se desea particionar. Se utiliza la directiva *#pragma omp parallel for* para paralelizar el bucle que recorre el arreglo.
- **Función swap:** esta función se utiliza para intercambiar dos elementos en el arreglo arr. Toma dos argumentos: dos punteros a enteros a y b que indican las posiciones de los elementos a intercambiar.
- **Función main:** esta función es la función principal del programa. Primero se declara un arreglo de enteros arr de tamaño n y se inicializa con números aleatorios entre 0 y 99 utilizando la función rand() de la biblioteca stdlib.h. Luego se mide el tiempo de ejecución utilizando las funciones *omp_get_wtime()*. Se utiliza la directiva **#pragma omp parallel** para crear un equipo de hilos y la directiva **#pragma omp single** para ejecutar el algoritmo quicksort en un solo hilo.

2.2. Merge Sort

Los módulos del Merge Sort son los siguientes:

- **Bibliotecas:** el código comienza importando las bibliotecas necesarias para su funcionamiento: stdio.h, stdlib.h, omp.h, string.h y time.h. La biblioteca stdio.h se utiliza para entrada/salida estándar, stdlib.h para funciones generales, omp.h para el uso de OpenMP, string.h para copiar caracteres y time.h para la medición del tiempo de ejecución.
- **Función Merge:** implementa el algoritmo de fusión para ordenar un arreglo de ”n.[el]ementos en orden ascendente. El algoritmo de fusión es un algoritmo de ordenamiento recursivo que divide el arreglo original en dos mitades, las ordena por separado y luego fusiona las dos mitades ordenadas en un arreglo ordenado.
- **Función MergeSort:** es una implementación paralela del algoritmo de ordenamiento de fusión. La función recibe tres argumentos: el arreglo .[a]rray”que se va a ordenar, el número de elementos ”n.[en] el arreglo y un arreglo temporal ”temp”del mismo tamaño que .[a]rray”para almacenar los valores ordenados. Se utiliza la directiva *#pragma omp task firstprivate* para dividir una tarea en subtareas más pequeñas y paralelizar su ejecución. La directiva *firstprivate* se utiliza para especificar que una variable debe ser inicializada con su valor en la sección de origen antes de ser usada en cada tarea, se asegura que cada tarea reciba su propio valor inicial y no sea afectada por cambios en la sección de origen.
- **Función Main:** esta función es la función principal del programa. Primero se declara un arreglo de enteros data de tamaño n y se inicializa con números aleatorios entre 0 y 99 utilizando la función rand(). Luego se mide el tiempo de ejecución utilizando las funciones *omp_get_wtime()*. Se

utiliza la directiva **#pragma omp parallel** para crear un grupo de hilos y la directiva **#pragma omp single** para ejecutar el algoritmo MergeSort en un solo hilo.

Análisis y Resultados

2.3. Quick Sort

2.3.1. Programa Serial: Quick Sort

Número de valores	Tiempo (s)
1,000	0,000000
10,000	0,005000
100,000	0,147000
200,000	0,443000
300,000	0,934000

Tabla 1: Serial Quick sort

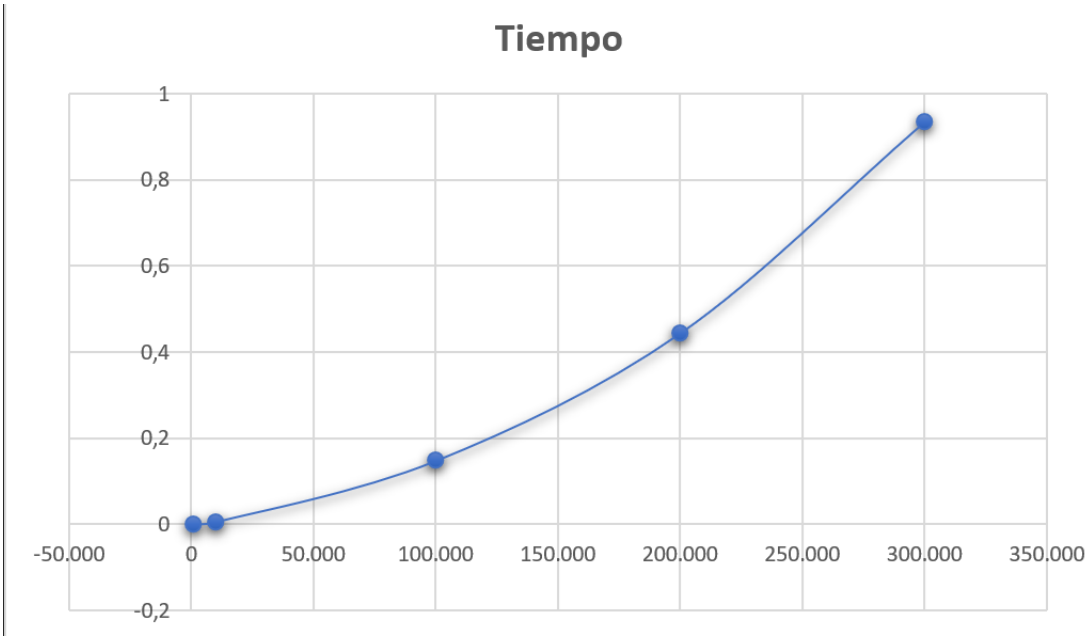

Figura 5: Análisis de resultados del programa QuickSort en programación paralelo

2.3.2. Programa Paralelo: Quick Sort

$1,000$ elementos	
Número de hilos	Tiempo
2	0,009000
4	0,008000
8	0,007000
16	0,008000
32	0,013000

$10,000$ elementos	
Número de hilos	Tiempo
2	0,055000
4	0,053000
8	0,051000
16	0,055000
32	0,056000

$100,000$ elementos	
Número de hilos	Tiempo
2	0,576000
4	0,556000
8	0,543000
16	0,523000
32	0,544000

$200,000$ elementos	
Número de hilos	Tiempo
2	1,331000
4	1,365000
8	1,374000
16	1,400000
32	1,392000

Tabla 2: Comparación tiempo Parallel Quick sort

2.4. Merge Sort

1,000 elementos	
Número de hilos	Tiempo
2	0,00300002
4	0,00500011
8	0,00800014
16	0,0119998
32	0,0180001

10,000 elementos	
Número de hilos	Tiempo
2	0,013
4	0,0140002
8	0,0220001
16	0,0409999
32	0,0379999

100,000 elementos	
Número de hilos	Tiempo
2	0,099
4	0,142
8	0,19
16	0,238
32	0,233

200,000 elementos	
Número de hilos	Tiempo
2	0,232
4	0,31
8	0,375
16	0,455
32	0,452

Tabla 3: Comparación tiempo Parallel Merge sort

Códigos Fuente De Los Programas

2.5. QuickSort

2.5.1. Serial en lenguaje C

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define n 10000

void swap(int *a, int *b);
int partition(int arr[], int low, int high);
void quicksort(int arr[], int low, int high);

int main() {
    int i, j;
    int arr[n];

    srand(time(0));

    for (j = 0; j < n; j++){
      arr[j] = rand() % 100;
    }

     double start_time = clock();

    quicksort(arr, 0, n - 1);

    double end_time = clock();

    printf("Arreglo ordenado: ");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");

    double time = ((double) (end_time - start_time)) / CLOCKS_PER_SEC;

    printf("\nNumero de elementos a ordenar: %d\n", n);

    printf("\nTiempo de ejecuci n: %f segundos\n", time);
    return 0;
}

void quicksort(int arr[], int low, int high) {
    if (low < high) {
        int pivotIndex = partition(arr, low, high);

        quicksort(arr, low, pivotIndex - 1);
```

```
        quicksort(arr, pivotIndex + 1, high);
    }
}

void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}
```

Listing 1: QuickSort en C

2.5.2. Paralelo en OMP

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <time.h>
#define n 10000

void quicksort(int arr[], int low, int high);
int partition(int arr[], int low, int high);
void swap(int* a, int* b);

int main() {
    int arr[n];
    int i, j;

     srand(time(0));

    for (j = 0; j < n; j++){
      arr[j] = rand() % 100;
    }

    double start_time = omp_get_wtime();
```

```
22
23      #pragma omp parallel num_threads(10)
24      {
25          #pragma omp single
26          quicksort(arr, 0, n - 1);
27      }
28
29      double end_time = omp_get_wtime();
30
31      printf("Arreglo ordenado: ");
32      for (i = 0; i < n; i++)
33          printf("%d ", arr[i]);
34      printf("\n");
35
36      printf("\nTiempo de ejecuci n: %f segundos\n", end_time - start_time);
37      return 0;
38  }
39
40  void quicksort(int arr[], int low, int high) {
41      if (low < high) {
42          int pivotIndex = partition(arr, low, high);
43
44          #pragma omp parallel sections
45          {
46               #pragma omp section
47              quicksort(arr, low, pivotIndex - 1);
48               #pragma omp section
49              quicksort(arr, pivotIndex + 1, high);
50          }
51      }
52  }
53
54  void swap(int* a, int* b) {
55      int temp = *a;
56      *a = *b;
57      *b = temp;
58  }
59
60  int partition(int arr[], int low, int high) {
61      int pivot = arr[high];
62      int i = (low - 1);
63
64      for (int j = low; j <= high - 1; j++) {
65          if (arr[j] < pivot) {
66              i++;
67              swap(&arr[i], &arr[j]);
68          }
69      }
70      swap(&arr[i + 1], &arr[high]);
```

```
    return (i + 1);
}
```

Listing 2: QuickSort en OMP

2.6. MergeSort

2.6.1. Serial en lenguaje C

```
#include <stdio.h>   //bibliotecas necesarias para el programa, que son stdio.h
   para las funciones de entrada/salida,
#include <stdlib.h>  //stdlib.h para funciones de memoria din mica y
#include <time.h>    //time.h para la generaci n de n meros aleatorios.

// Funci n para mezclar dos subarreglos ordenados de un arreglo dado
void merge(int arr[], int l, int m, int r) {
    int i, j, k;
    int n1 = m - l + 1;  // Tama o del subarreglo izquierdo
    int n2 = r - m;      // Tama o del subarreglo derecho

    int L[n1], R[n2];    // Arreglos temporales para almacenar los subarreglos

    // Copia los elementos del subarreglo izquierdo al arreglo temporal L[]
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    // Copia los elementos del subarreglo derecho al arreglo temporal R[]
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    i = 0;
    j = 0;
    k = l;

    // Mezcla los subarreglos ordenadamente en el arreglo original
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    // Copia los elementos restantes del subarreglo izquierdo en el arreglo
   original
```

```
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    // Copia los elementos restantes del subarreglo derecho en el arreglo original
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

// Funci n para ordenar un arreglo utilizando Merge Sort
void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;  // Calcula el  ndice  medio del arreglo

        // Ordena recursivamente las dos mitades del arreglo
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        // Mezcla las dos mitades ordenadas del arreglo
        merge(arr, l, m, r);
    }
}

int main() {
    int n = 10;
    int arr[n];

    // Genera n meros aleatorios en el arreglo utilizando srand() y rand()
    srand(time(NULL));
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 100;  // Genera un n mero aleatorio entre 0 y 99
    }

    // Imprime el arreglo original
    printf("Arreglo original aleatorio: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    // Ordena el arreglo utilizando Merge Sort
    mergeSort(arr, 0, n - 1);

    // Imprime el arreglo ordenado
```

```
    printf("\nArreglo ordenado: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

Listing 3: MergeSort en C

2.6.2. Paralelo en OMP

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


void Merge(int * array, int n, int * temp);
void MergeSort(int * array, int n, int * temp);

int main()
{
   int n = 1000;
   double start_time, end_time;

   int data[n], temp[n];

   srand(time(NULL));

   for (int i = 0; i < n; i++)
   {
      data[i]=rand() %1000;
   }


   printf("Arreglo antes de ordenamiento\n");
   for (int i = 0; i < n; i++)
   {
      printf("%d  ",data[i]);
   }

   start_time = omp_get_wtime();

   #pragma omp parallel num_threads(100)
   {
      #pragma omp single
      MergeSort(data, n, temp);
```

```
38    }
39    end_time = omp_get_wtime();
40
41    printf("\nArreglo ordenado\n");
42
43    for (int i = 0; i < n; i++)
44    {
45       printf("%d  ",data[i]);
46    }
47
48    printf("\nNumero de elementos a ordenar: %d",n);
49    printf("\nTiempo de ejecucion: %g segundos\n",end_time-start_time);
50
51    return 0;
52 }
53
54 void Merge(int * array, int n, int * temp) {
55    int a = 0;
56    int b = n/2;
57    int ti = 0;
58   //a itera hasta la primera mitad y b itera la segunda mitad del arreglo
59    while (a<n/2 && b<n) {
60       if (array[a] < array[b]) {
61          temp[ti] = array[a];
62          ti++;
63          a++;
64       } else {
65          temp[ti] = array[b];
66          ti++;
67         b++;
68       }
69    }
70    while (a<n/2) { //primera mitad
71       temp[ti] = array[a];
72      ti++;
73      a++;
74    }
75    while (b<n) { //segunda mitad
76       temp[ti] = array[b];
77       ti++;
78      b++;
79    }
80
81    memcpy(array, temp, n*sizeof(int));    //copia el arreglo ordenado en el
      arreglo original
82
83 }
84
85
```

```
86 void MergeSort(int * array, int n, int * temp)
87 {
88    if (n < 2) return;
89
90    #pragma omp task firstprivate (array, n, temp)
91    MergeSort(array, n/2, temp);
92
93    #pragma omp task firstprivate (array, n, temp)
94    MergeSort(array+(n/2), n-(n/2), temp);
95
96
97    #pragma omp taskwait                    //directiva que espera que ambas
       tareas terminen su ejecucion
98
99
100   Merge(array, n, temp);                  //fusion de los arreglos ordenados
101 }
```

Listing 4: MergeSort en OMP

Descripción de los resultados visualizados en el monitor

2.7. Quick Sort:

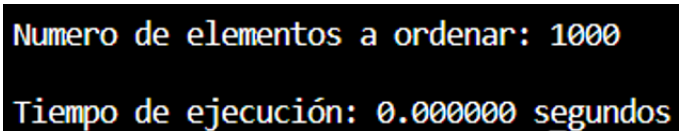

Figura 6: Quicksort serial con n=1,000

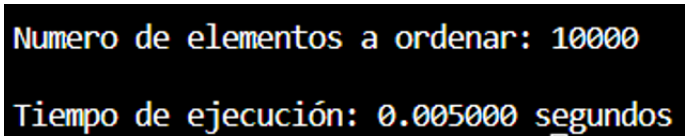

Figura 7: Quicksort serial con n=10,000

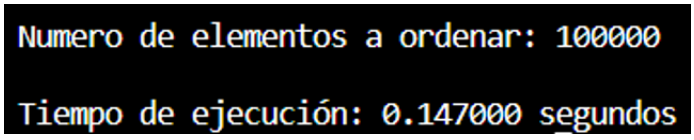

Figura 8: Quicksort serial con n=100,000

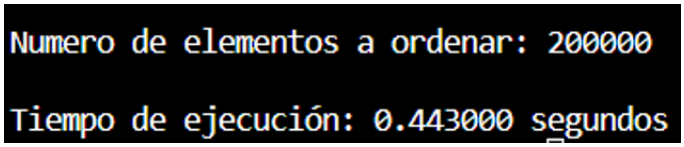

Figura 9: Quicksort serial con n=200,000

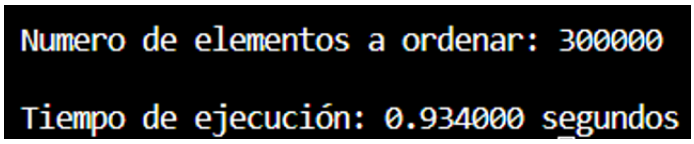

Figura 10: Quicksort serial con n=300,000

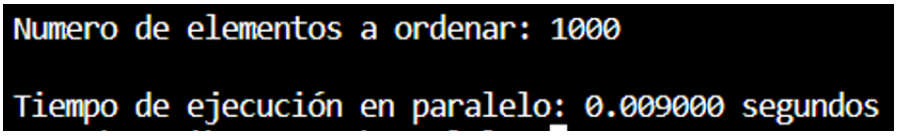

Figura 11: Quicksort paralelo con 2 hilos y 1000 elementos

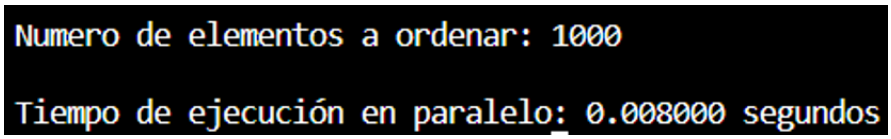

Figura 12: Quicksort paralelo con 4 hilos y 1000 elementos

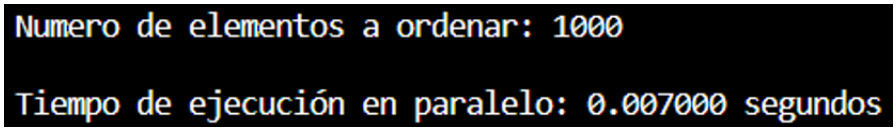

Figura 13: Quicksrt paralelo con 8 hilos y 1000 elementos

2.8. Merge Sort:

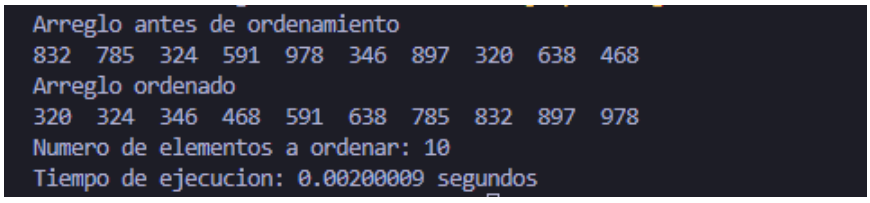

Figura 14: Merge sort paralelo con 2 hilos y 10 elementos

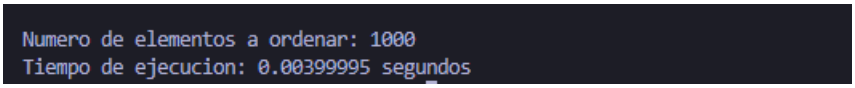

Figura 15: Merge sort paralelo con 2 hilos y 1000 elementos

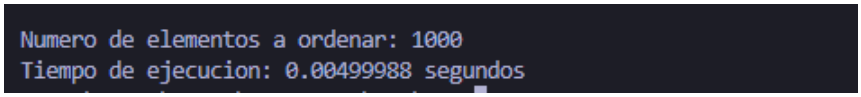

Figura 16: Merge sort paralelo con 4 hilos y 1000 elementos

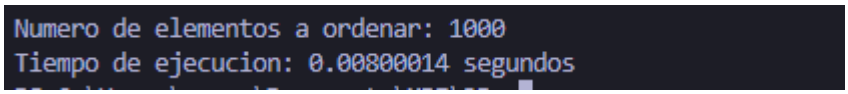

Figura 17: Merge sort paralelo con 8 hilos y 1000 elementos

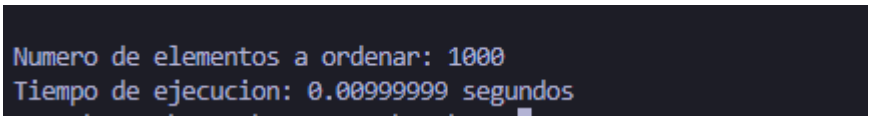

Figura 18: Merge sort paralelo con 16 hilos y 1000 elementos

Diagramas y documentos adicionales

2.9. Diagrama de Quick sort:

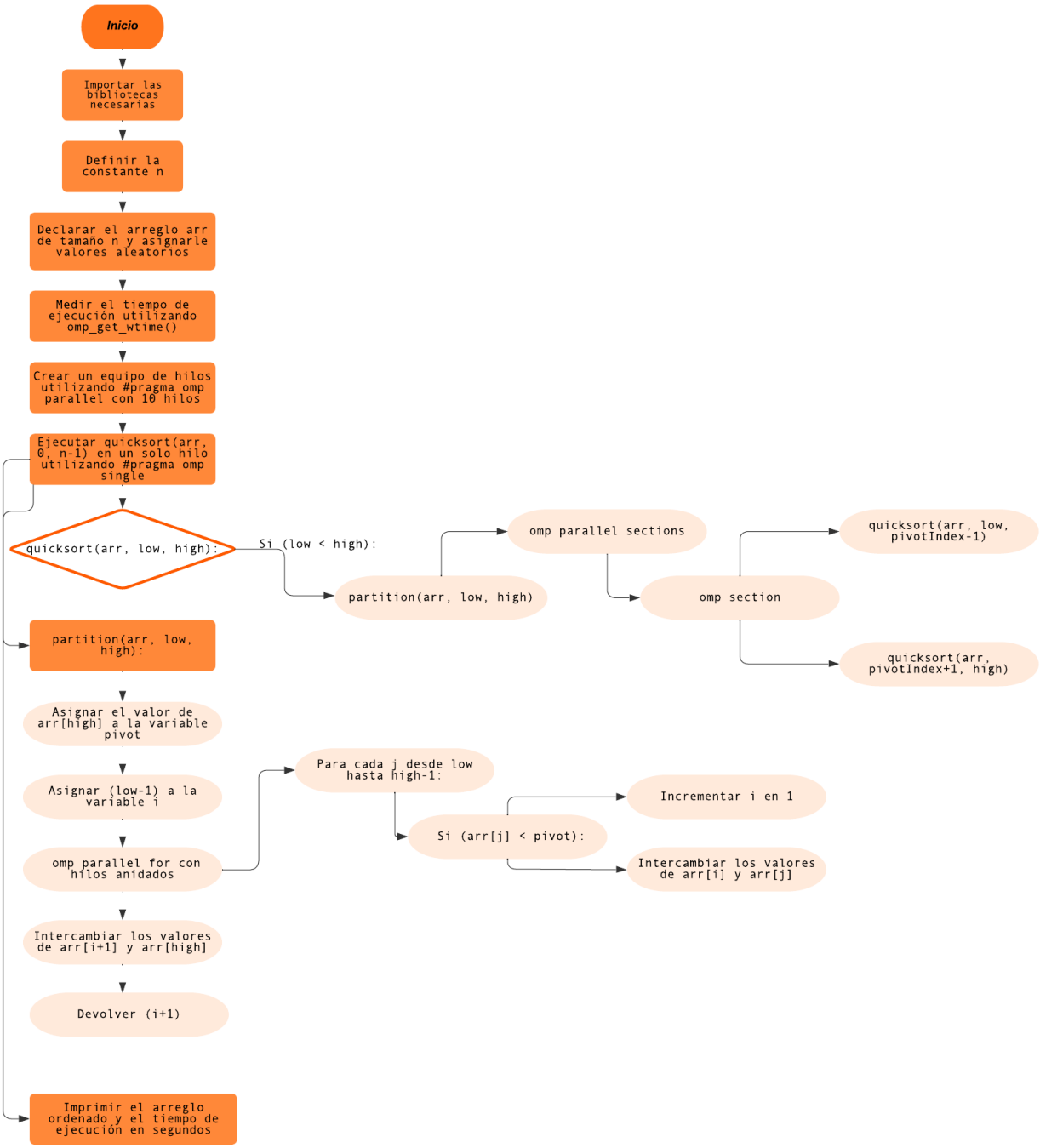

Figura 19: Diagrama de flujo Quicksort

2.10. Diagrama de Merge sort:

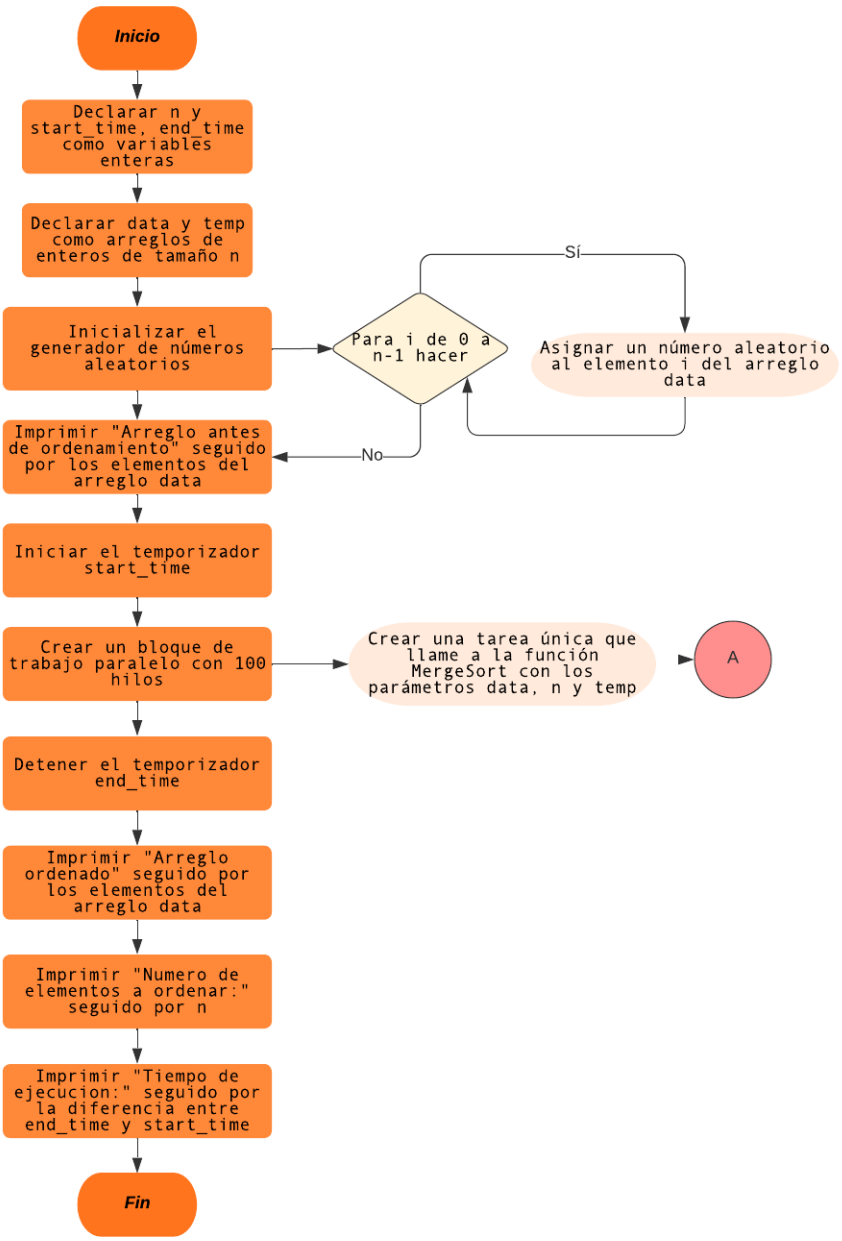

Figura 20: Diagrama de flujo Mergesort main()

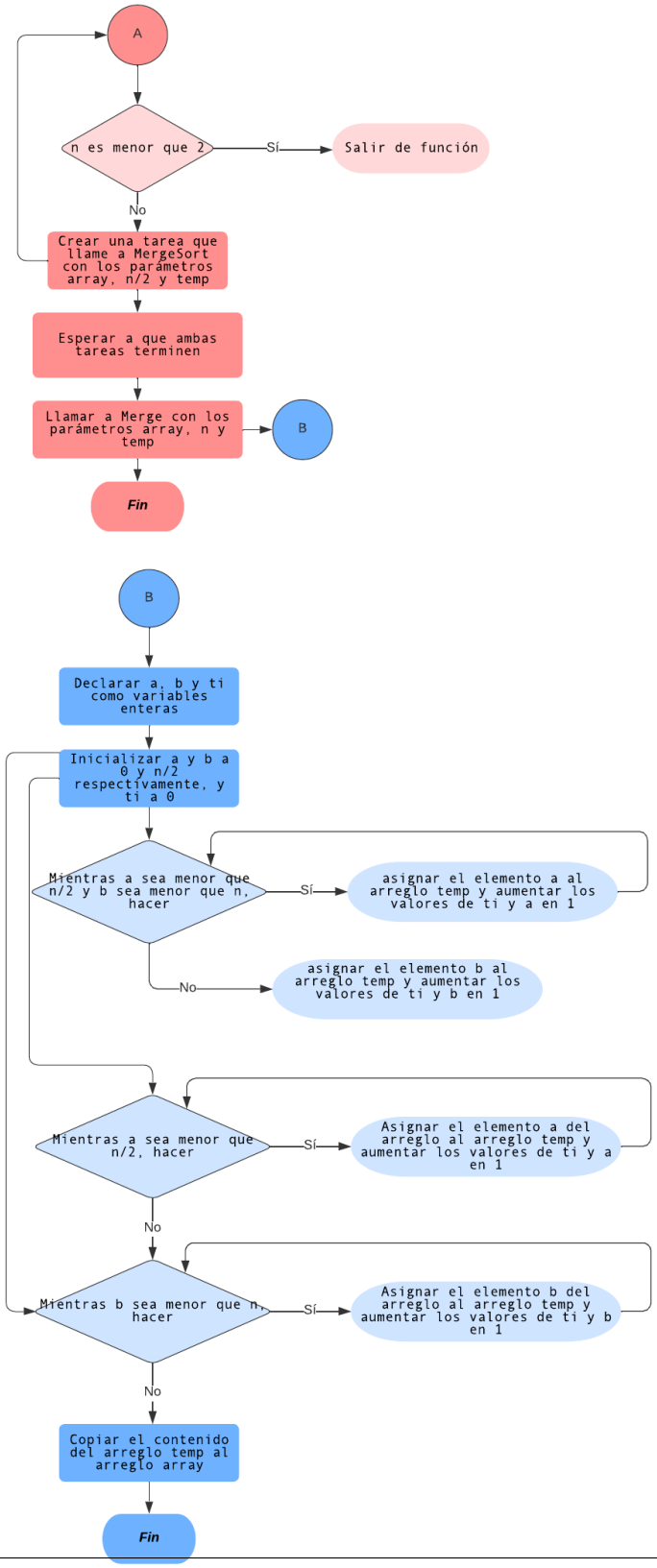

Figura 21: Diagrama de flujo Mergesort funciones

Conclusión

En general, es mejor paralelizar algoritmos de ordenamiento porque esto puede permitir que el proceso de ordenamiento se realice de manera mucho más rápida y eficiente en comparación con un algoritmo de ordenamiento secuencial. Al dividir el trabajo de ordenamiento en varias tareas que se realizan simultáneamente en diferentes núcleos de procesamiento, se puede reducir significativamente el tiempo total necesario para completar la tarea. Además, el uso de la paralelización también puede permitir que se procesen conjuntos de datos más grandes, lo que puede ser crítico para aplicaciones que manejan grandes volúmenes de información. Sin embargo, es importante tener en cuenta que la paralelización no siempre es la mejor opción para todas las situaciones y puede requerir una consideración cuidadosa de las necesidades y limitaciones específicas de cada caso en particular.