Comparación de estrategias de paralelización para Merge Sort

Ramirez Victor, Ruiz Joaquin, Olivares Agustín
Facultad de Ingeniería
Universidad Nacional de Cuyo

Abstract: En este documento se analizan distintas estrategias de paralelización del algoritmo Merge Sort utilizando MPI. Se estudian enfoques como la asignación de sublistas individuales a nodos, la división equitativa de la lista según la cantidad de nodos disponibles y la recombinación de sublistas en paralelo.

Keywords: Merge Sort, Eficiencia, Algoritmos de Ordenamiento, SpeedUp

1. Introducción

1.1. Merge Sort

El algoritmo Merge Sort es una técnica de ordenamiento eficiente que se basa en el paradigma de divide y vencerás. Su funcionamiento consiste en dividir repetidamente la lista a ordenar en sublistas más pequeñas hasta que cada una contenga uno o ningún elemento, ya que en ese caso se consideran trivialmente ordenadas. Luego, estas sublistas se combinan (o fusionan) en forma ordenada, reconstruyendo progresivamente la lista original pero en orden creciente.

Conceptualmente, Merge Sort sigue cuatro pasos fundamentales:

1. Si la lista contiene cero o un elemento, se considera ordenada y no se realiza ninguna acción.
2. Si la lista tiene más de un elemento, se divide en dos sublistas de tamaño aproximadamente igual.
3. Cada sublista se ordena de forma recursiva aplicando nuevamente el algoritmo Merge Sort.
4. Finalmente, se combinan las dos sublistas ordenadas en una única lista también ordenada.

Este enfoque asegura una complejidad temporal de $O(nlogn)$ en todos los casos (mejor, peor y promedio), lo que convierte a Merge Sort en una opción robusta para ordenar grandes volúmenes de datos, especialmente cuando se requiere estabilidad en el ordenamiento.

2. Desarrollo

2.1. Algoritmo secuencial

La versión secuencial del algoritmo *Merge Sort* fue implementada en lenguaje C, haciendo uso de recursividad para dividir el arreglo original y de una función auxiliar para realizar la recombinación ordenada de las sublistas. El objetivo de esta implementación es ordenar un arreglo de números enteros generados aleatoriamente y medir el tiempo que tarda el proceso para su posterior comparación con la implementación paralela del algoritmo.

El algoritmo comienza por verificar que se haya proporcionado correctamente el tamaño del arreglo como argumento en la línea de comandos. Luego, se asigna memoria dinámica para el arreglo y se lo llena con valores enteros aleatorios comprendidos entre 1 y 10.000, utilizando la función *rand()* y la semilla basada en el tiempo actual *(srand(time(NULL)))*.

El núcleo del algoritmo lo componen dos funciones principales:

1. **mergeSort**: Es una función recursiva que aplica el principio de *divide y vencerás*. Si el segmento del arreglo contiene más de un elemento, se calcula el punto medio y se divide el arreglo en dos mitades. Luego se llama recursivamente a sí misma para ordenar cada sublista por separado. Finalmente, se llama a la función merge para fusionar ambas mitades ya ordenadas.

2. **merge**: Esta función se encarga de combinar dos subarreglos previamente ordenados. Para ello, se crean dos arreglos auxiliares dinámicos (uno para cada mitad), y se copian los datos desde el arreglo original. Posteriormente, se realiza la fusión de los elementos comparando sus valores uno a uno, y almacenándolos en la posición correspondiente del arreglo original. Finalmente, se liberan los arreglos auxiliares mediante free() para evitar fugas de memoria.

La eficiencia del algoritmo es medida utilizando la función clock() de la biblioteca <time.h>, la cual permite calcular el tiempo transcurrido entre el inicio y el fin del proceso de ordenamiento.

Pseudocódigo de nuestra implementación:

```
Funcion MergeSort(arreglo, izquierda, derecha)
    Si izquierda < derecha entonces
        medio ← (izquierda + derecha) / 2
        MergeSort(arreglo, izquierda, medio)
        MergeSort(arreglo, medio + 1, derecha)
        Merge(arreglo, izquierda, medio, derecha)
    FinSi
FinFuncion

Funcion Merge(arreglo, izquierda, medio, derecha)
    Crear subarreglos izquierdo y derecho
    Combinar ambos subarreglos en orden ascendente
    Insertar resultado en la posición correspondiente del arreglo original
FinFuncion

Inicio
    Leer cantidad de elementos n
    Generar arreglo de tamaño n con valores aleatorios
    Llamar a MergeSort(arreglo, 0, n - 1)
Fin
```

2.2. Algoritmo Paralelo

2.2.1. Primera versión: Asignación de sublistas individuales a nodos

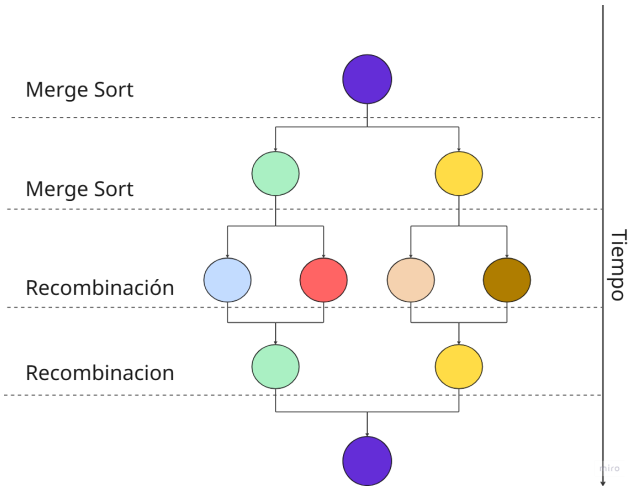

Figura 1: Asignación directa de sublistas a nodos

En esta primera versión, la idea inicial fue aplicar el algoritmo Merge Sort dividiendo recursivamente la lista original y asignar cada sublista generada a un nodo computacional diferente para que se encargue de ordenarla. Sin embargo, este enfoque resultó ser ineficiente. A medida que se aplica la recursión del Merge Sort, la cantidad de sublistas crece exponencialmente. Por ejemplo, si el algoritmo genera 30 sublistas, se requerirían 30 nodos para procesarlas en paralelo, lo cual no es escalable ni viable en entornos con recursos computacionales limitados. Esta estrategia no contempla una distribución eficiente del trabajo, ya que presupone que existe un nodo disponible por cada sublista, lo cual rápidamente se vuelve inviable.

2.2.2. Segunda versión: División equitativa de la lista según la cantidad de nodos disponibles

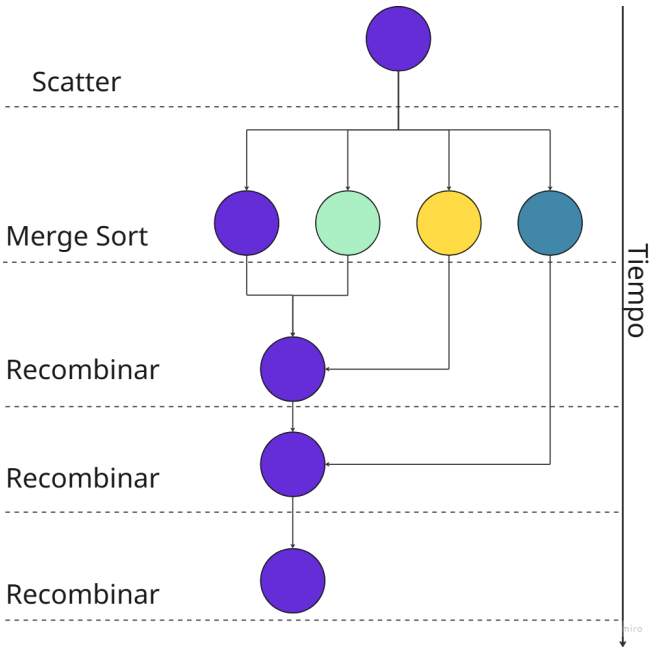

Figura 2: División equitativa con recombinación secuencial

En la segunda versión, se propone un enfoque más controlado y práctico. La lista original se divide en tantas partes como nodos computacionales estén disponibles, utilizando la función MPI_Scatter para distribuir equitativamente los fragmentos entre los nodos. Luego, cada nodo aplica el algoritmo Merge Sort de manera local sobre su sublista. Una vez ordenadas, las sublistas son recolectadas por el nodo maestro mediante MPI_Gather. Posteriormente, el nodo maestro se encarga de recombinar las sublistas ordenadas para reconstruir la lista completa de forma ordenada.

El principal inconveniente de este enfoque es que la fase de recombinación recae exclusivamente en el nodo maestro. Como se observa en la Figura 2, el nodo maestro debe fusionar primero las dos primeras sublistas, luego combinar el resultado con la tercera sublista y finalmente con la cuarta. Esto implica que, tras completar la ordenación local, los nodos trabajadores quedan inactivos mientras el nodo maestro realiza toda la recombinación, generando un cuello de botella y desaprovechando el paralelismo disponible.

2.2.3. Tercera versión: Recombinación de sublistas en paralelo

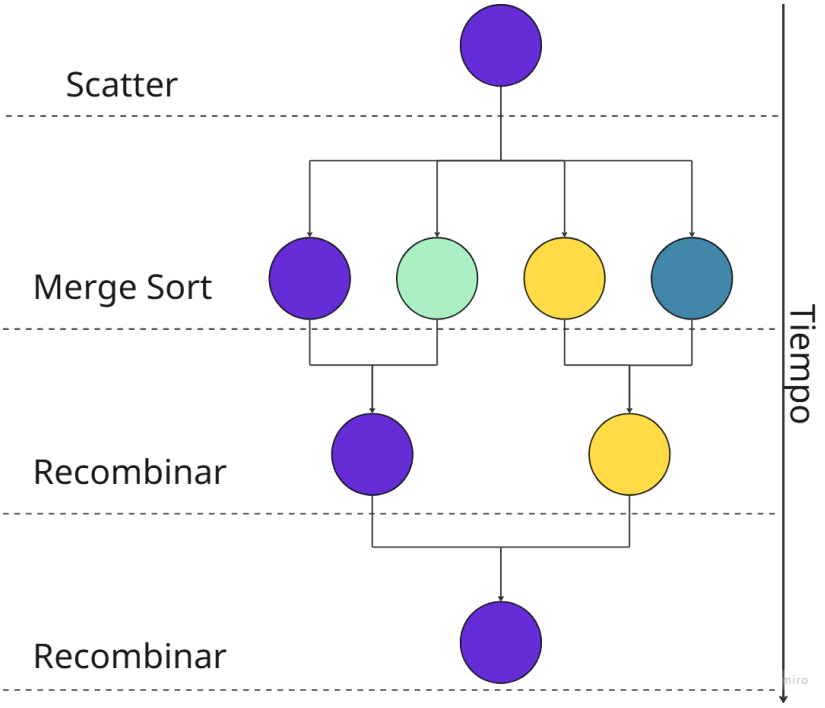

Figura 3: Recombinación de sublistas en paralelo

La tercera versión busca resolver el problema de la recombinación secuencial centralizada en el nodo maestro. En este enfoque, la fusión de sublistas ordenadas se realiza de forma paralela, distribuyendo el trabajo entre varios nodos. De esta manera, no solo el nodo maestro participa en la fase de recombinación, sino que también los nodos trabajadores contribuyen en distintas etapas del proceso de fusión. Esta estrategia permite una mejor utilización de los recursos disponibles, reduce la carga del nodo maestro y mejora significativamente la eficiencia global del algoritmo paralelo. Al distribuir la recombinación, se reduce el tiempo de ejecución total y se evita el cuello de botella que se presentaba en la versión anterior.

Pseudo código para la versión 3:

```
INICIO

Inicializar MPI
Obtener rank y size

# donde n es la cantidad de elementos
SI argumentos inválidos o n <= 0
    Si soy el maestro, mostrar error
    Finalizar

Dividir los n elementos entre los procesos (counts y displs)

SI soy el maestro
    Crear lista aleatoria de n elementos

Distribuir los datos entre procesos (MPI_Scatterv)

Cada proceso:
    Ordena su parte local con mergeSort
```

```
# Combinación entre pares
Mientras step < size:
    Si rank es múltiplo de 2*step
        Recibe datos del proceso rank + step y los fusiona
    Sino
        Envía datos al proceso rank - step y termina

Si soy el maestro
    # (Opcional: mostrar lista ordenada)
    Mostrar tiempo de ejecución

Finalizar MPI

FIN
```

3. Diseño de Experimentos

Para el diseño experimental de nuestro algoritmo, optamos por implementar y evaluar la tercera versión en un entorno paralelo, ya que ofrece el mejor equilibrio entre eficiencia y *speedup* del sistema. Consideramos que esta versión se adapta de forma más adecuada al paradigma de programación paralela, lo que permite aprovechar de manera óptima los recursos disponibles, en particular los nodos de procesamiento, durante la ejecución del algoritmo y la resolución del problema.

Para llevar a cabo la experimentación, se definieron diferentes configuraciones de entrada y entornos de ejecución, permitiendo evaluar el comportamiento del programa bajo distintas condiciones. Las pruebas se realizaron variando tanto el tamaño de los arreglos a ordenar como la cantidad de procesos utilizados en la versión paralela mediante MPI (Message Passing Interface). De esta manera, fue posible analizar la escalabilidad del algoritmo y su desempeño relativo frente a la implementación secuencial.

Parámetros y variables consideradas

- **Tamaño de entrada**: Se utilizarán arreglos con tamaños crecientes (por ejemplo: 1×10^4, 1×10^5, 1×10^6 elementos), con números generados aleatoriamente entre 1 y 10000.

- **Cantidad de procesos (MPI)**: En la versión paralela, se probarán distintas configuraciones con 2, 4, 8 y 16 procesos para observar la evolución del rendimiento con mayor paralelismo.

- **Tiempos de ejecución**: Se medirá el tiempo total de ejecución de cada versión, utilizando la librería <time.h>.

- **Speedup**: Se calculará el *speedup* como el cociente entre el tiempo de ejecución secuencial y el paralelo.

- **Eficiencia**: También se evaluará la eficiencia paralela, definida como.