Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencias de la Computación
CC7315 - Programación Multicore

Challenge 3: Comparison of Parallel Sorting Algorithms

Diego Madariaga Román

1. Descripción

Este desafío consiste en la comparación de 4 algoritmos paralelos para el ordenamiento de números enteros.

A continuación, se presentarán los algoritmos utilizados en el proceso de experimentación y los resultados obtenidos, los cuales son finalmente analizados.

2. Experimentación

El código utilizado y las instrucciones para reproducir los experimentos realizados se encuentran disponibles en `https://git.io/dmadariaga-sorting`

Para los experimentos realizados se utilizaron 3 algoritmos de ordenamiento vistos en clase[1]: `parallel_quick_sort.cpp`, `parallel_merge_sort.cpp` y `parallel_counting_sort.c`, en donde los dos primeros perteneces a la categoría de algoritmos de ordenamiento comparativos. Además, se implementó una versión del algoritmo *MSD Radix Sort* (Most Significant Digit), el cual se encuentra al igual que *Counting Sort* en la categoría de algoritmos de ordenamiento no comparativos. Dicho algoritmo funciona de la siguiente manera:

1. Toma el bit más significativo de cada número
2. Agrupa a los números en 2 casilleros (*buckets*) según su bit más significativo
3. Recursivamente ordena cada *bucket* tomando el siguiente bit más significativo
4. Retorna la concatenación de los 2 casilleros creados

Este algoritmo es fácilmente paralelizable[2], ejecutando en paralelo las dos llamadas recursivas del método. Así, se obtiene el cuarto algoritmo para comparar: `parallel_radix_sort.cpp`

[1] `https://github.com/jfuentess/multicore-programming/tree/master/Code/fork-join`
[2] `https://software.intel.com/sites/default/files/m/d/4/1/d/8/Akki_RadixSort.pdf`

Con cada algoritmo de ordenamiento, se realizó el ordenamiento de arreglos de números enteros de tamaño $\{10^4, 10^5, 10^6, 10^7, 10^8\}$ en alfabetos de tamaño $\{2^8, 2^{11}, 2^{15}, 2^{20}\}$, utilizando desde 1 a 12 procesadores y midiendo el tiempo de ejecución y el número de *cache-misses*.

Cada experimento fue realizado 3 veces y se han reportado los valores correspondientes a las medianas obtenidas en cada caso.

Todos los experimentos fueron llevados a cabo utilizando un computador de 12 cores físicos, distribuidos en una arquitectura NUMA de 2 nodos, cada uno con 6 cores y 3GB de memoria RAM. Los resultados se muestran a continuación:

2.1. Tiempos de ejecución

2.1.1. Tamaño del arreglo a ordenar: 10^6

Figura 2.1: Tiempo en [s] para el ordenamiento de 10^6 números enteros en un alfabeto de tamaño 2^8.

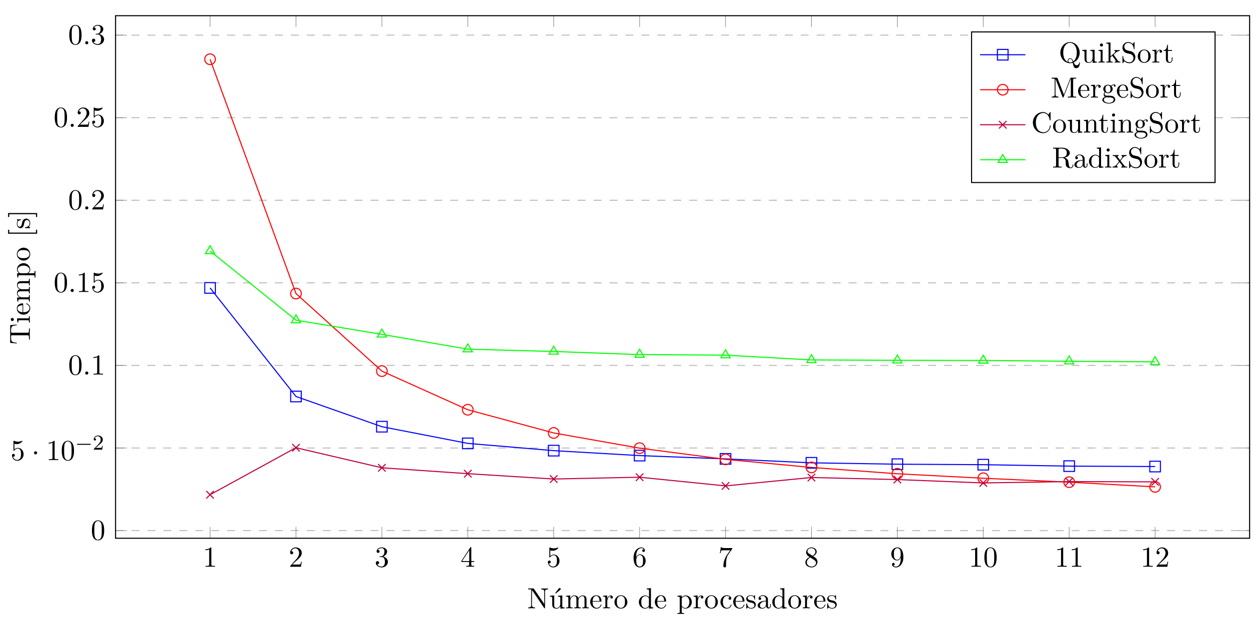


Figura 2.2: Tiempo en [s] para el ordenamiento de 10^6 números enteros en un alfabeto de tamaño 2^{15}.

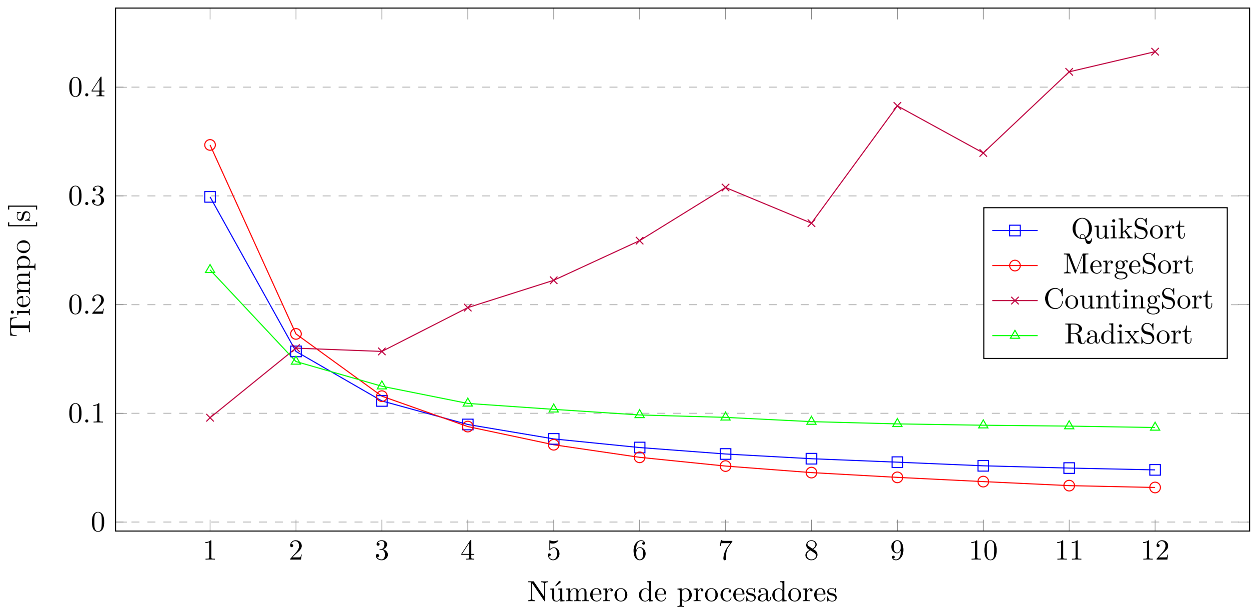
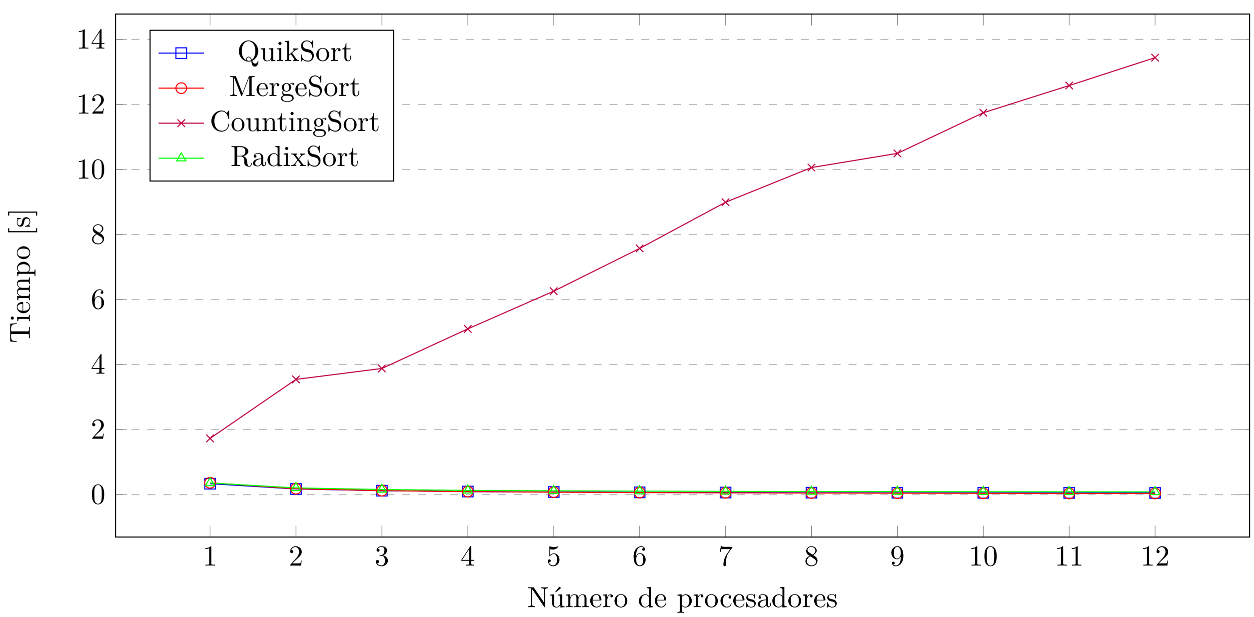


Figura 2.3: Tiempo en [s] para el ordenamiento de 10^6 números enteros en un alfabeto de tamaño 2^{20}.



2.1.2. Tamaño del arreglo a ordenar: 10^7

Figura 2.4: Tiempo en [s] para el ordenamiento de 10^7 números enteros en un alfabeto de tamaño 2^8.

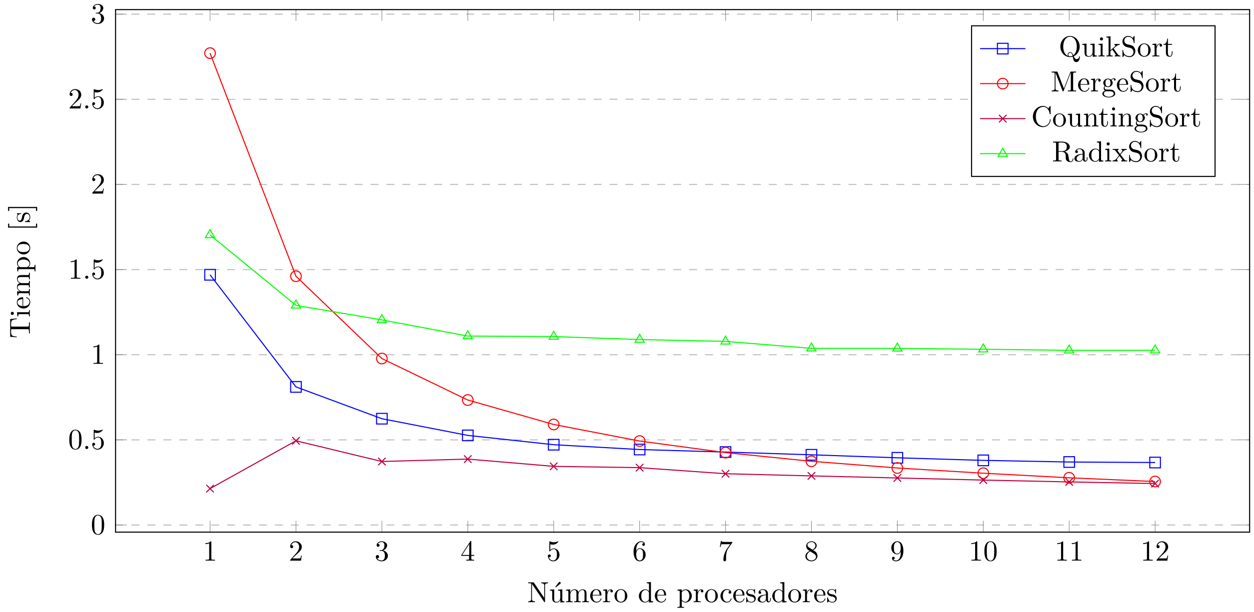


Figura 2.5: Tiempo en [s] para el ordenamiento de 10^7 números enteros en un alfabeto de tamaño 2^{15}.

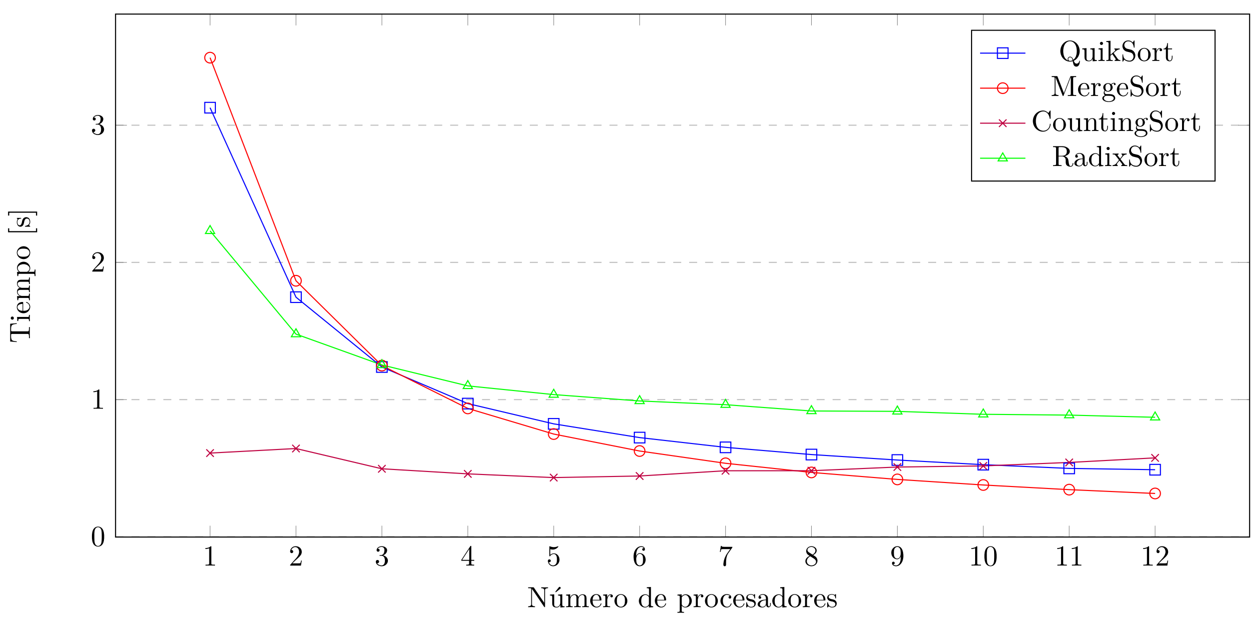
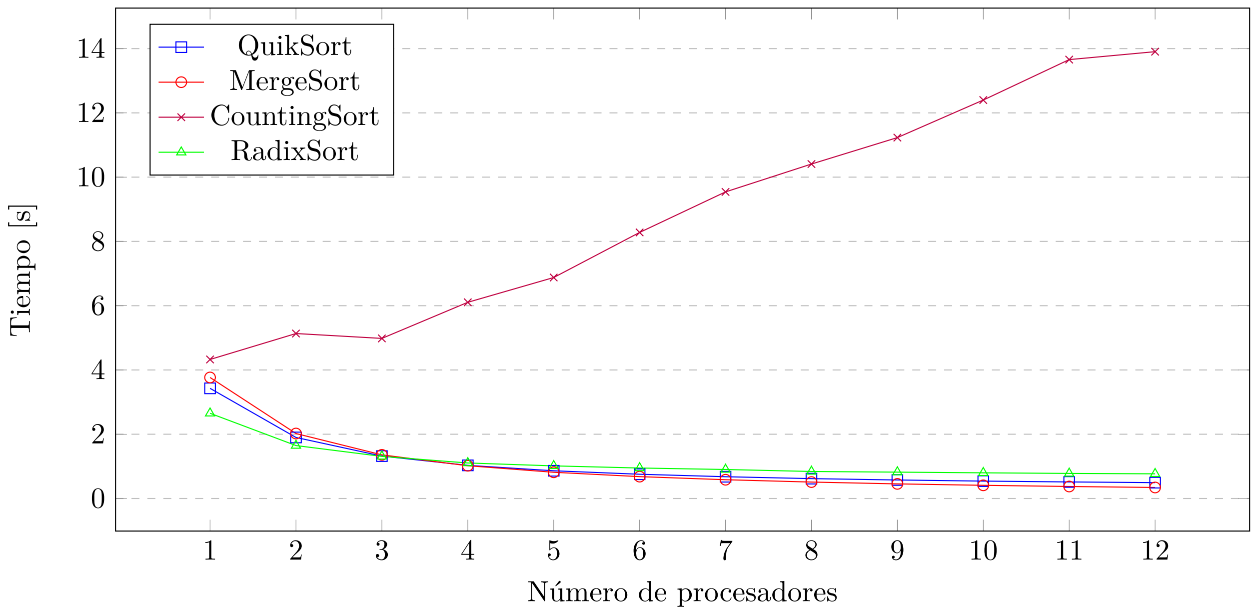


Figura 2.6: Tiempo en [s] para el ordenamiento de 10^7 números enteros en un alfabeto de tamaño 2^{20}.



2.1.3. Tamaño del arreglo a ordenar: 10^8

Figura 2.7: Tiempo en [s] para el ordenamiento de 10^8 números enteros en un alfabeto de tamaño 2^8.

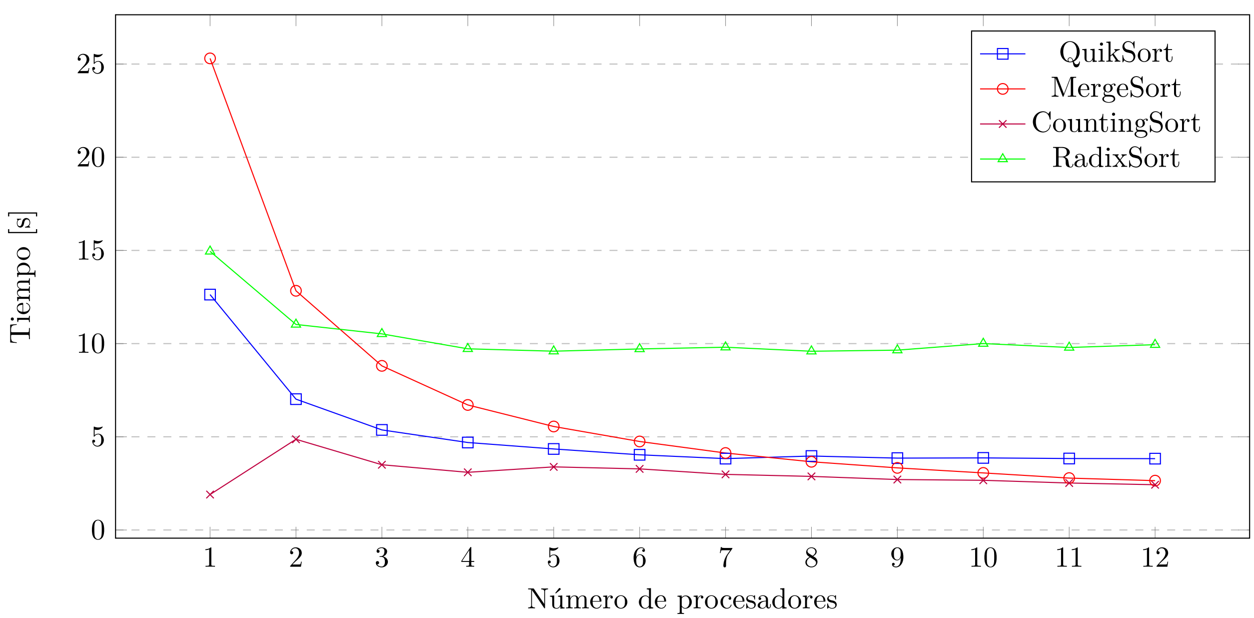


Figura 2.8: Tiempo en [s] para el ordenamiento de 10^8 números enteros en un alfabeto de tamaño 2^{15}.

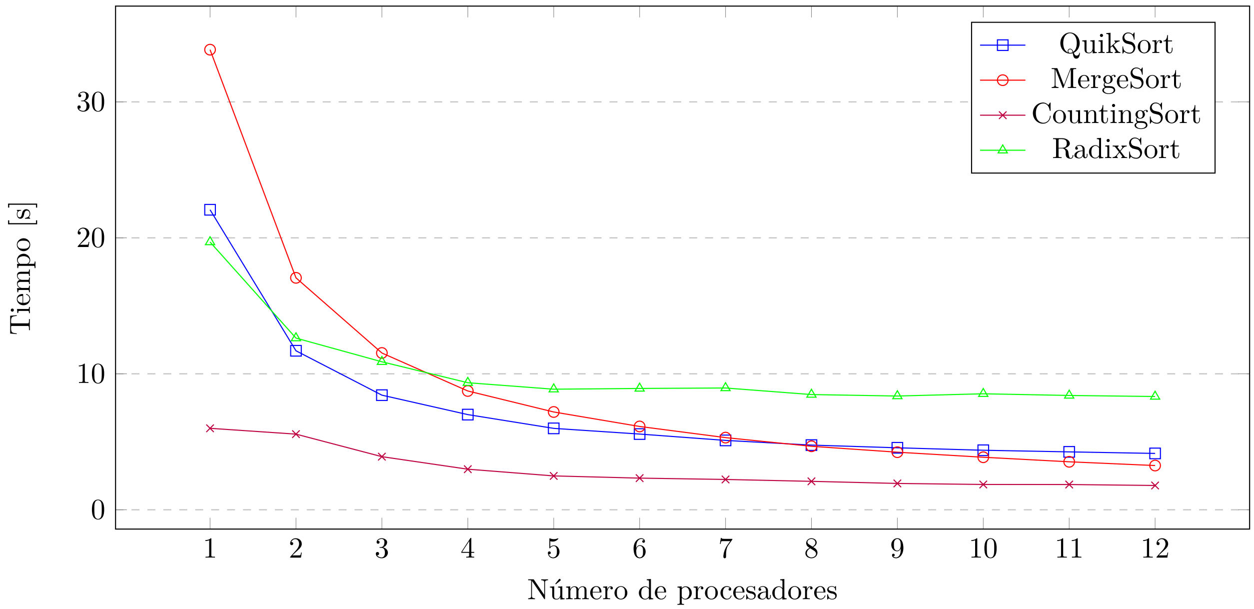
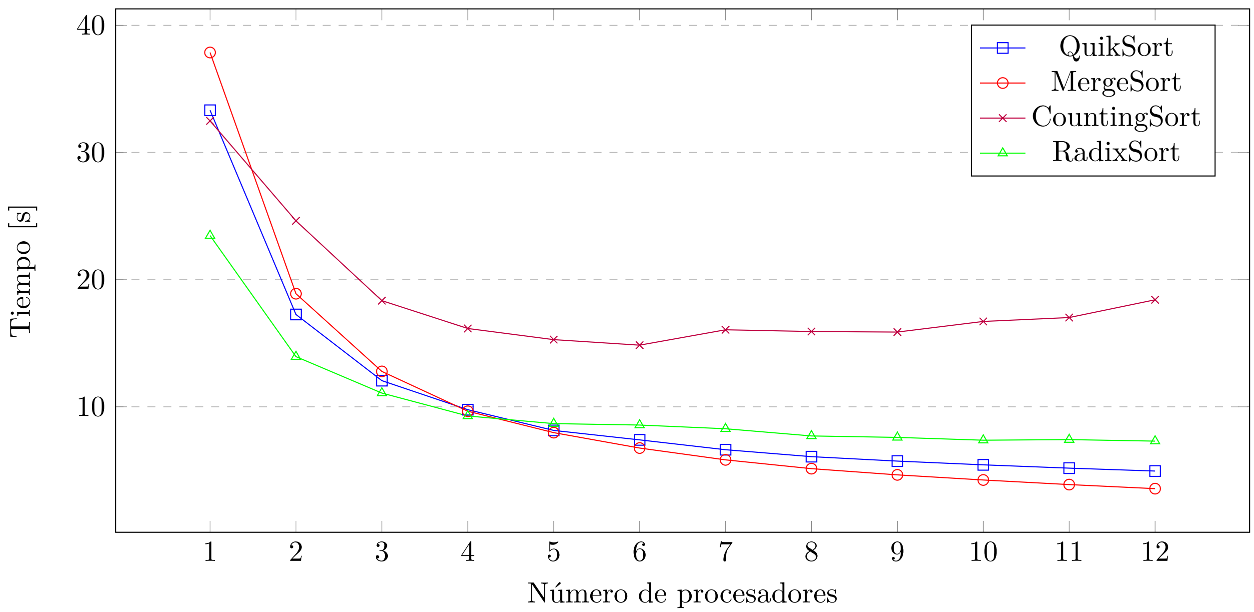


Figura 2.9: Tiempo en [s] para el ordenamiento de 10^8 números enteros en un alfabeto de tamaño 2^{20}.



2.2. *Cache-misses*

2.2.1. Tamaño del arreglo a ordenar: 10^6

Figura 2.10: *Cache-misses* para el ordenamiento de 10^6 números enteros en un alfabeto de tamaño 2^8.

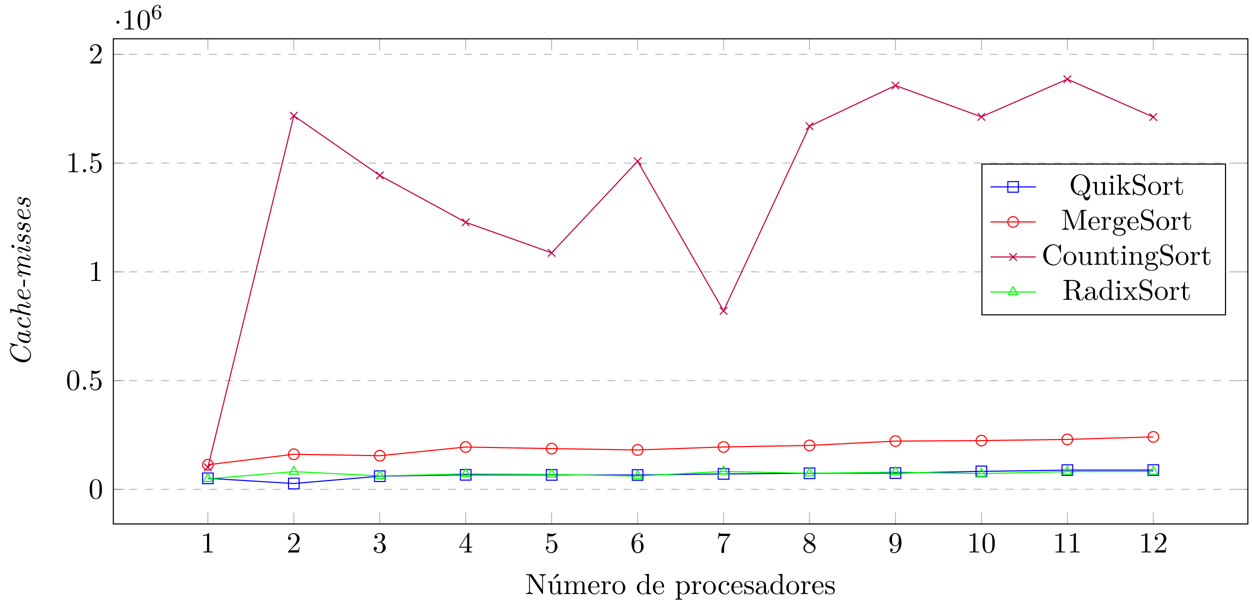


Figura 2.11: *Cache-misses* para el ordenamiento de 10^6 números enteros en un alfabeto de tamaño 2^{15}.

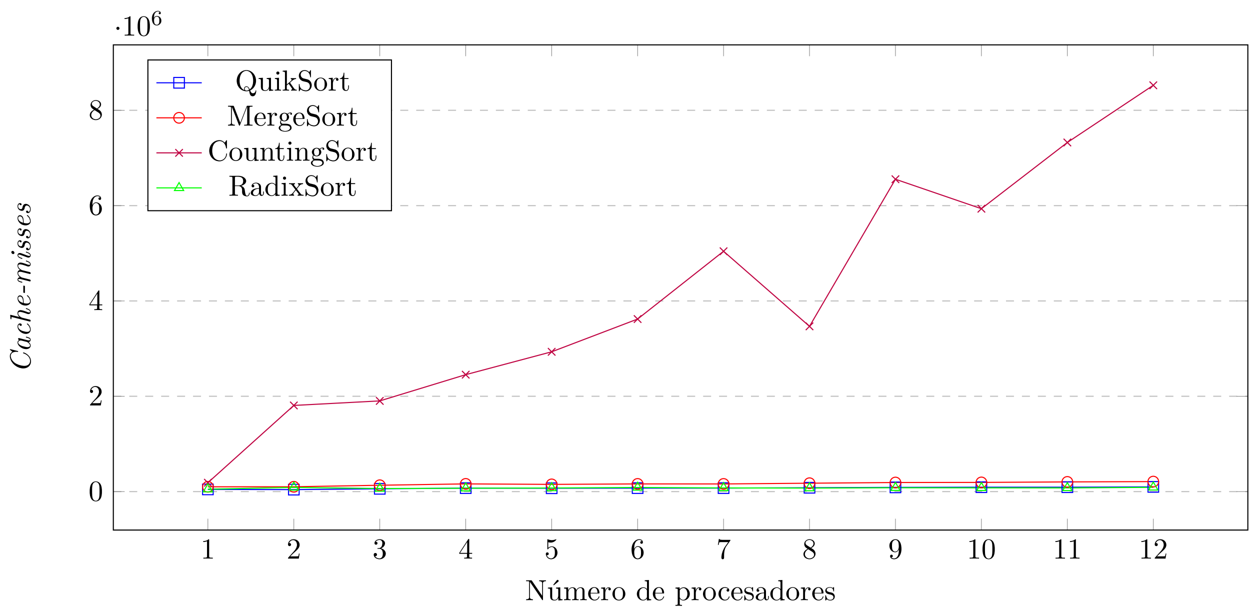
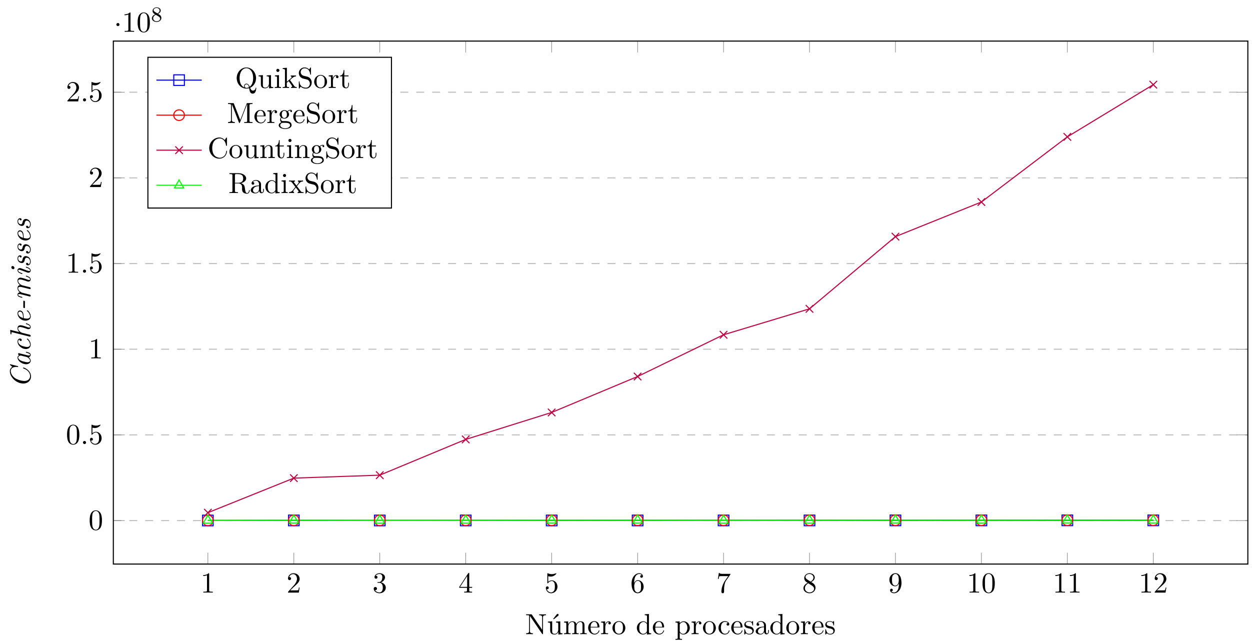


Figura 2.12: *Cache-misses* para el ordenamiento de 10^6 números enteros en un alfabeto de tamaño 2^{20}.



2.2.2. Tamaño del arreglo a ordenar: 10^7

Figura 2.13: *Cache-misses* para el ordenamiento de 10^7 números enteros en un alfabeto de tamaño 2^8.

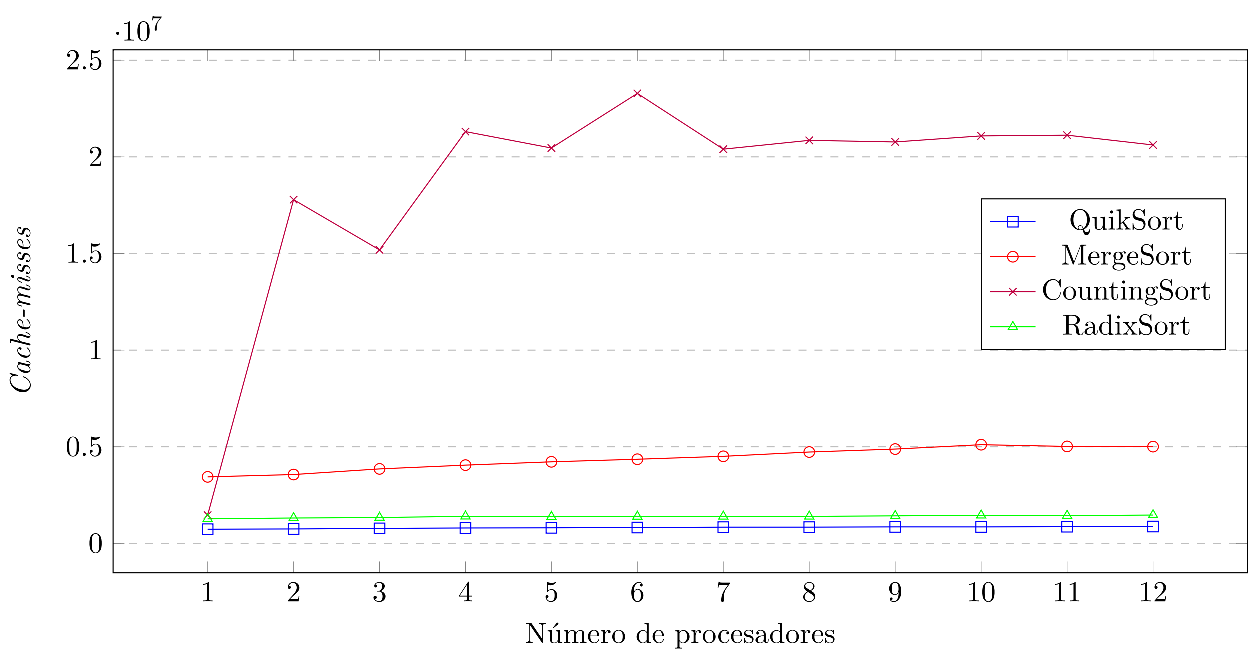


Figura 2.14: *Cache-misses* para el ordenamiento de 10^7 números enteros en un alfabeto de tamaño 2^{15}.

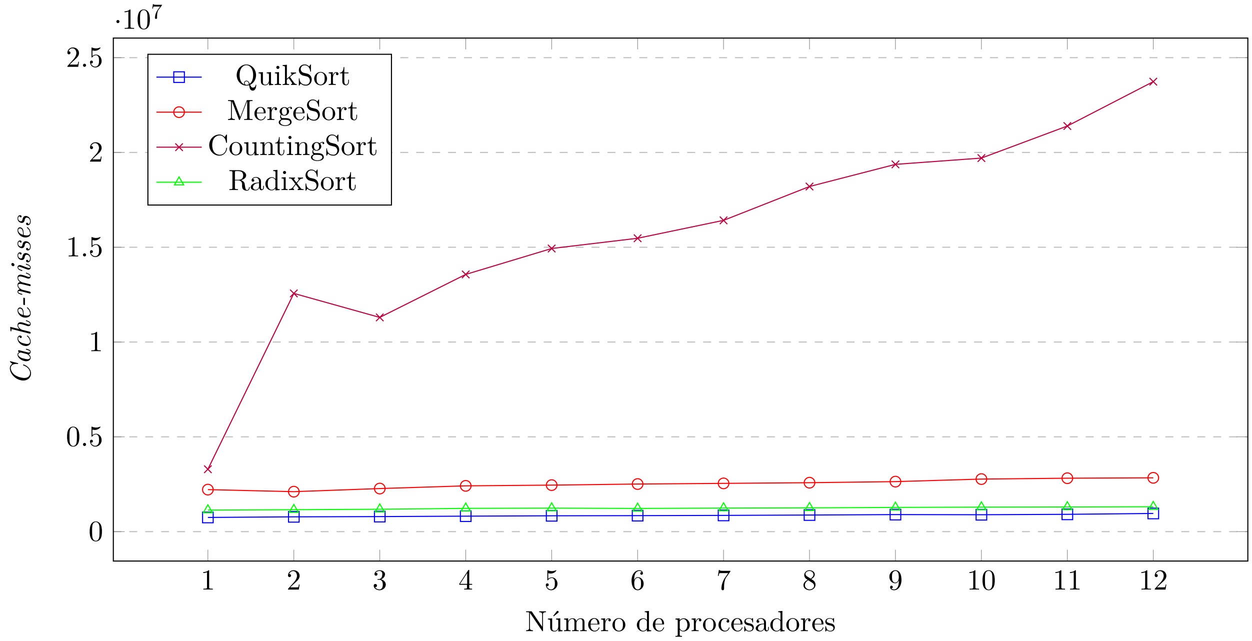
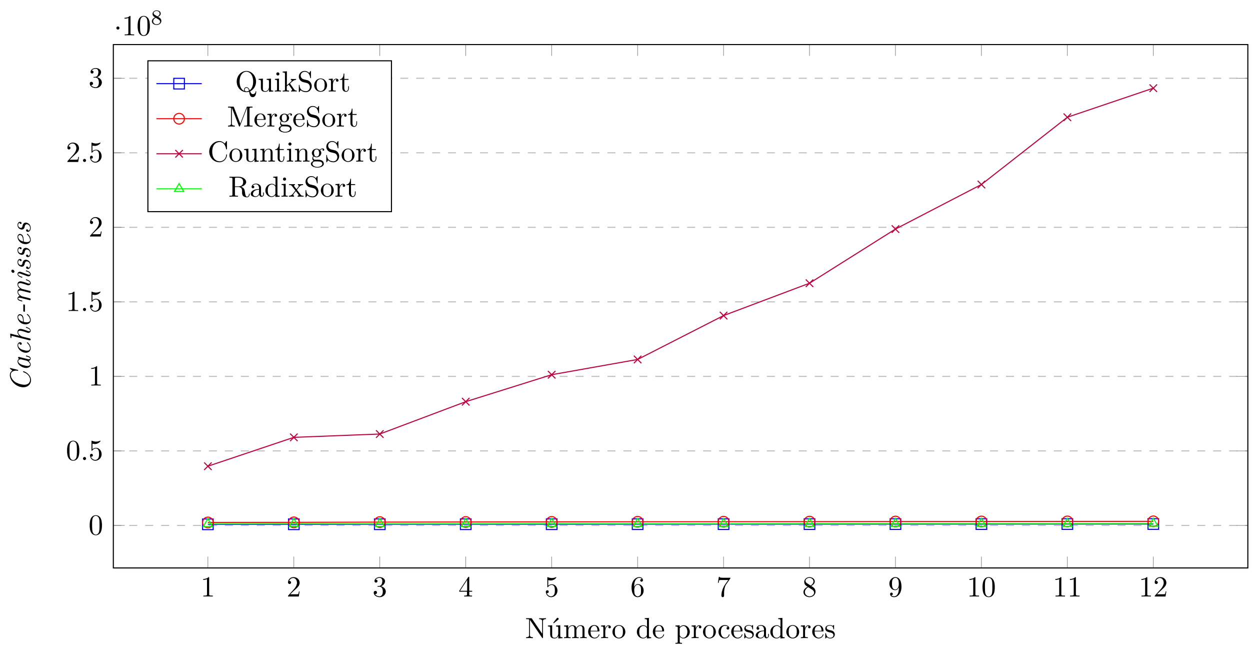


Figura 2.15: *Cache-misses* para el ordenamiento de 10^7 números enteros en un alfabeto de tamaño 2^{20}.



2.2.3. Tamaño del arreglo a ordenar: 10^8

Figura 2.16: *Cache-misses* para el ordenamiento de 10^8 números enteros en un alfabeto de tamaño 2^8.

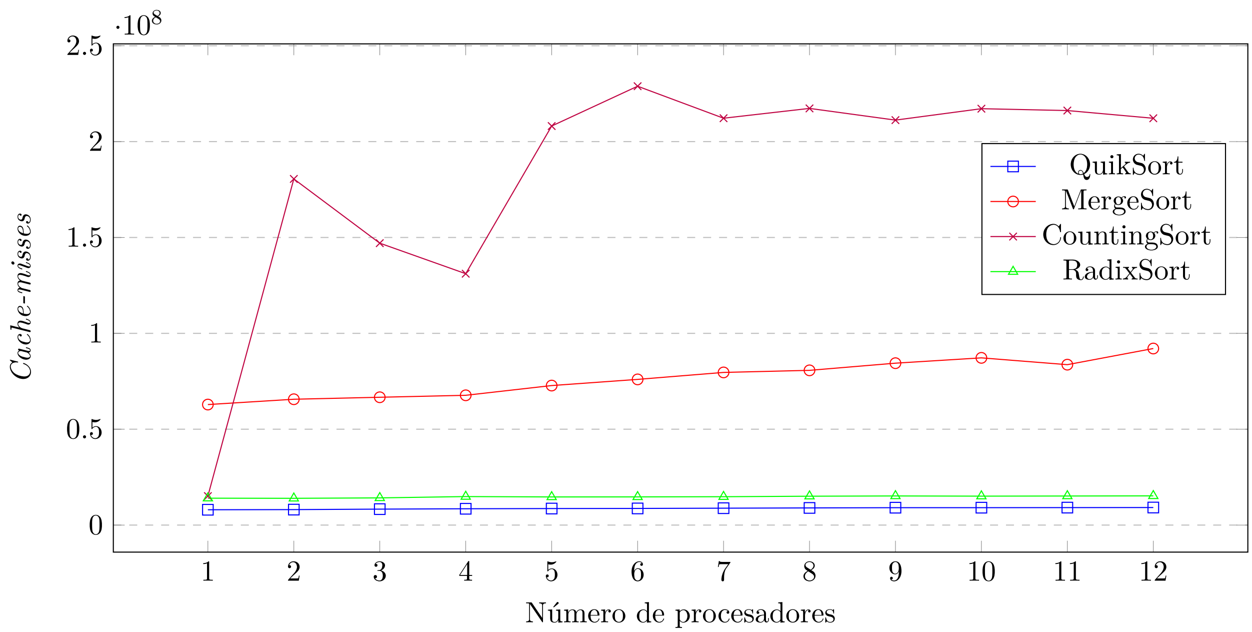


Figura 2.17: *Cache-misses* para el ordenamiento de 10^8 números enteros en un alfabeto de tamaño 2^{15}.

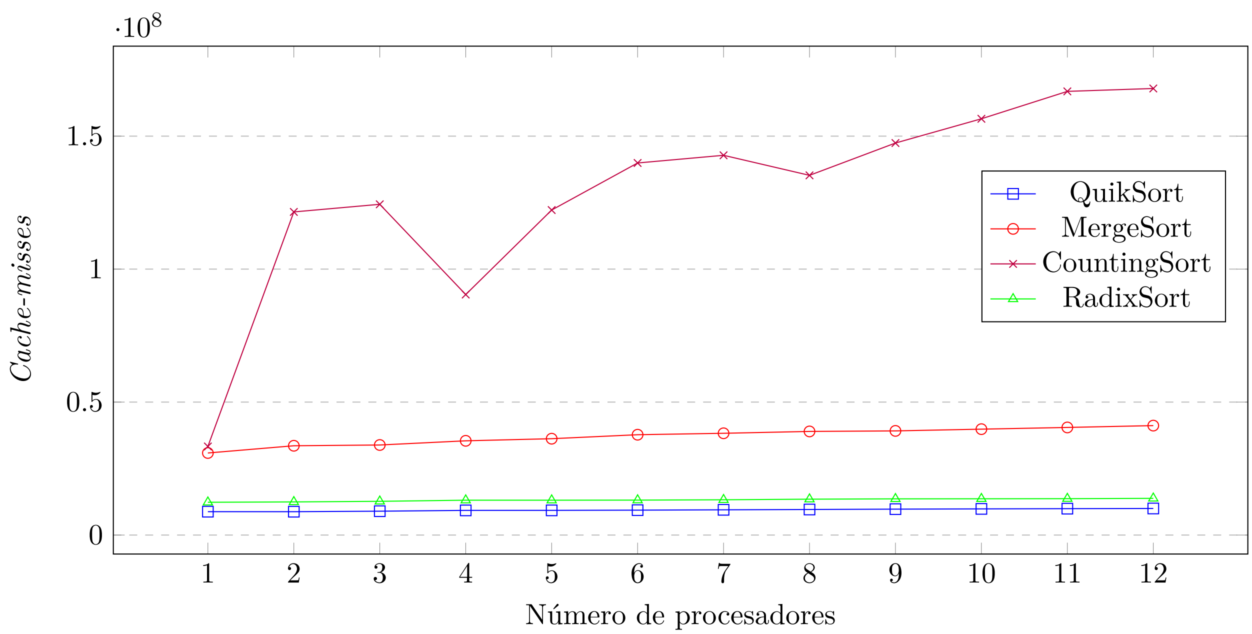
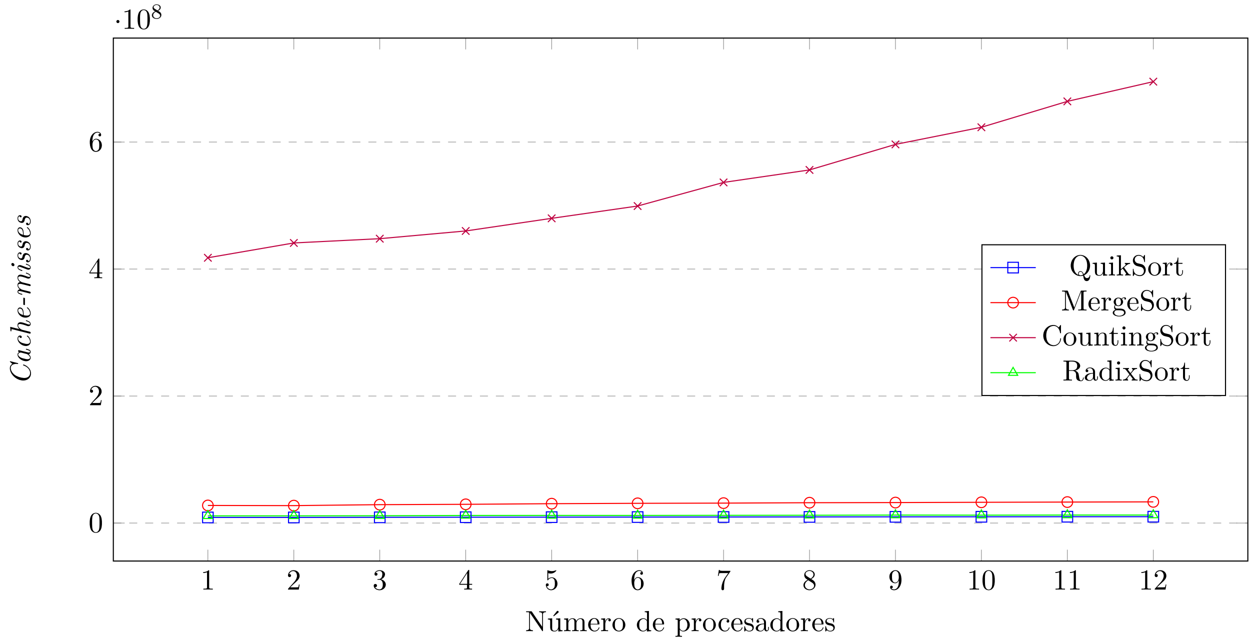


Figura 2.18: *Cache-misses* para el ordenamiento de 10^8 números enteros en un alfabeto de tamaño 2^{20}.



3. Discusión y Conclusiones

Dado los resultados anteriores, es posible decir que el algoritmo *CountingSort* presenta buenos resultados de tiempos de ejecución cuando el tamaño del alfabeto es considerablemente menor al tamaño del arreglo a ordenar (Figuras 2.1, 2.4, 2.5, 2.7, 2.8), obteniendo tiempos mejores que los otros 3 algoritmos, utlizando de 1 a 12 procesadores. Sin embargo, cuando disminuye la brecha entre dichos valores, los tiempos de ejecución empeoran rápidamente, alejándose notoriamente de los otros 3 algoritmos (Figuras 2.2, 2.3, 2.6, 2.9). Por lo tanto, para alfabetos de tamaños pequeños, *CountingSort* se muestra como la mejor alternativa.

En cuanto al algoritmo *RadixSort*, podemos observar que para un arreglo de tamaño 10^6 (Figuras 2.1, 2.2 y 2.3) a medida que aumenta el tamaño del alfabeto, se obtienen tiempos de ejecución similares a los obtenidos por los algoritmos *QuickSort* y *MergeSort*. El mismo comportamiento descrito se puede observar para tamaños de entrada 10^7 (Figuras 2.4, 2.5 y 2.6) y 10^8 (Figuras 2.7, 2.8 y 2.9). Es importante notar que para números altos del tamaño de la entrada y del alfabeto, al utilizar un número de procesadores entre 1 y 4, *RadixSort* obtiene mejores tiempos de ejecución que los otros 3 algoritmos (Figuras 2.5, 2.6, 2.9). Dado lo anterior, si se cuenta con una máquina de 4 o menos procesadores físicos, *RadixSort* es una muy buena alternativa a utilizar.

Los algoritmos *QuickSort* y *MergeSort* presentan los comportamientos más estables con respecto al tamaño del arreglo de entrada y al tamaño del alfabeto (Figuras 2.1-2.9). Se puede observar que a pesar de presentar tiempos muy similares entre ellos, se cumple de forma general que para un número de prcesadores cercano a 1, *QuickSort* tiene menores tiempos de ejecución, mientras que para un número de procesadores cercano a 12, *MergeSort* obtiene menores tiempos. Ambos algoritmos se muestran como una buena alternativa para uso general.

Con respecto al número de *cache-misses*, las Figuras 2.10-2.18 muestran que *CountingSort* genera un número muy alto de *cache-misses*, el cual aumenta con el número de procesadores utilizados, incluso en los experimentos donde *CountingSort* tiene los menores tiempos de ejecución. Se observa también, que los otros 3 algoritmos presentan un número de *cache-misses* constante en función del número de procesadores, en donde el algoritmo *MergeSort* presenta un valor ligeramente más alto con respecto a *QuickSort* y *RadixSort*, ambos con una cantidad de *cache-misses* muy bajos.