COMPARACIÓN ENTRE ALGORITMOS DE ORDENAMIENTO PARALELIZADOS EN JAVA

COMPARISON BETWEEN SORTING ALGORITHMS PARALLELIZED IN JAVA

Diego Gómez García
Universidad De la Salle Bajío, México
dgg2602748@udelasalle.edu.mx

José Luís Alonzo Velázquez
Golstats, Departamento de R&D, México
pepe@cimat.mx

Recepción: 19/septiembre/2019 **Aceptación:** 11/noviembre/2019

Resumen

Se analiza el rendimiento en tiempo, de los algoritmos de ordenamiento Bubble Sort, Odd Even Sort, Rank Sort, Counting Sort, Radix Sort, Merge Sort, Bitonic Sort y Quick Sort; paralelizados y ejecutados en una instancia de la Máquina Virtual de Java (JVM). En el caso de Odd Even Sort, la sincronización al esperar en la barrera cíclica reduce la ventaja de procesamiento adquirida al incrementar la cantidad de núcleos. En cambio, Bubble Sort y Rank Sort, mejoran su rendimiento global al incrementar los núcleos y requerir menor sincronización.

Por otra parte, Bitonic Sort, Merge Sort y Quick Sort, tienen un tiempo de ejecución muy bajo al utilizarse en un solo núcleo, puede notarse que se obtiene un beneficio de múltiples núcleos, sin embargo, al acercarse al tiempo necesario en la JVM para lanzar los hilos, el beneficio de escalar la cantidad de núcleos deja de producir un beneficio.

Palabras Clave: comparación, multi-núcleo, ordenamiento.

Abstract

This analyses the time performance of sorting algorithms Bubble Sort, Odd Even Sort, Rank Sort, Counting Sort, Radix Sort, Merge Sort, Bitonic Sort and Quick Sort; parallelized and executed in an instance of the Java Virtual Machine (JVM). In the

case of Odd Even Sort, the wait at the synchronization cyclic barrier reduce the processing advantage acquired by incrementing the quantity of cores. In contrast, Bubble Sort and rank Sort, increase their global performance with the increment of number of cores and requiring less synchronization.

On the other hand, Bitonic Sort Merge Sort and Quick Sort, have a very low execution time when used in a single core, it can be noticed that a benefit of multiple cores, however, when approaching the necessary time in the JVM to launch threads, the benefit of scaling the number of cores stops producing a benefit.

***Keywords*:** comparison, *multi-core, sorting.*

1. Introducción

En los últimos años las mejoras en las técnicas de fabricación y diseño, así como los límites en la miniaturización de los componentes y la demanda de un mayor poder de cómputo, han empujado a la industria a la fabricación de CPU (Unidad Central de Procesamiento, CPU, por sus siglas en inglés) con múltiples núcleos, el uso eficiente de estas plataformas exige el estudio y desarrollo de algoritmos que puedan operar de forma paralela para resolver un problema haciendo uso así de cada núcleo de procesamiento disponible.

En el caso del lenguaje Java, cada aplicación está aislada y se ejecuta dentro de su propia instancia de máquina virtual [1, 1].

Un multiprocesador es un equipo constituido por dos o más microprocesadores, como es el caso de Tandy Corporation TRS-80 Model 16 que es uno de los primeros ejemplos de multiprocesador, dotado con un microprocesador principal y un Z80 como procesador de E/S, en cambio un procesador multi-núcleo tiene dos o más unidades de procesamiento independientes en un mismo componente.

Entre las ventajas del cómputo paralelo están el ahorro en tiempo y dinero, y son importantes para resolver problemas de mayor magnitud y complejidad, así como la posibilidad de utilizar recursos que no son locales, como la memoria RAM (Memoria de acceso aleatorio, RAM, por sus siglas en inglés) y CPU en otros nodos. Actualmente el cómputo paralelo se utiliza para modelar problemas en las áreas de la ciencia como la genética, química, geología y la ingeniería; en la industria y el

comercio con la minería de datos, la inteligencia artificial y el modelado económico, entre muchos otros. Un ejemplo de solución paralela es la aproximación de PI por el método Monte Carlo; este analiza una serie de puntos generados aleatoriamente del área de un cuadrado que contiene un círculo circunscrito, si un punto está dentro del círculo es considerado dentro del cálculo final, ya que se sabe la cantidad de puntos en total analizados, se puede determinar el valor de PI. Este método no tiene dependencia entre sus pasos, lo que lo convierte en un excelente proyecto para ser paralelizado, al separar la cantidad de puntos analizados por cada procesador y reduciendo el conteo al final. A diferencia del cálculo de PI, otros problemas tienen dependencia entre los resultados, se requiere comunicación entre los procesadores como es por ejemplo la resolución de una ecuación de calor en 2 dimensiones, donde cada punto en un plano discreto, es afectado por los puntos adyacentes. Este problema a primera vista es un arreglo de arreglos ya que las filas son un arreglo que a su vez contiene las columnas, por lo que paralelizarlo es trivial, sin embargo, el hecho de cada punto afecta a los adyacentes, provoca que los puntos en la frontera con los puntos asignados a otro procesador entren en conflicto al tratar de escribir al mismo tiempo una dirección de memoria, requiriendo que se comuniquen los procesadores para evitar esta condición de carrera [2].

La importancia y las aplicaciones del procesamiento paralelo, son ampliamente conocidas, especialmente los algoritmos de ordenamiento [3, p. 1]. En el caso específico de Java, *Quick Sort* es utilizado para ordenamiento de primitivos a través de *Arrays.sort()*, es relevante conocer la eficiencia real al ordenar elementos con características específicas, y así determinar si el método que provee Java en su API es la mejor opción para nuestro problema en particular.

Se plantea observar los efectos que tiene un intermediario en la ejecución del código como es la máquina virtual de java, que ejecuta código intermedio y que en su modelo la ejecución de hilos no garantiza su existencia en múltiples núcleos, sino que es administrado por la máquina virtual. Se implementaron los algoritmos descritos en "*Comparative Analysis of Sorting Methods using OpenMP*", reemplazando implementaciones por algunas otras en vista de poder utilizar el mismo arreglo de entrada [3, p. 1].

La sección 2 describe de manera general cada uno de los algoritmos de ordenamiento analizados en este trabajo, así como la adaptación del método de paralelización por buckets a cada uno de ellos. En la tercera sección se describen los resultados obtenidos de las versiones paralelizadas de cada uno de los algoritmos, así como la comparación entre ellos. En la sección 4 se presentan los resultados generales obtenidos en este trabajo.

2. Métodos

A continuación, se describen los métodos utilizados para el comparativo, en donde *p* representa el conjunto de núcleos, y *n* un elemento dentro del arreglo, p_i denota un núcleo en particular y de igual manera n_i un elemento en particular dentro del arreglo.

En la mayoría de los algoritmos se utilizó una optimización similar para su versión paralela, para *p* procesadores, se utilizó una barrera cíclica y un arreglo auxiliar del mismo tamaño del arreglo a ordenar, a cada p_i se le asigna un segmento del arreglo numérico para ser ordenado, al terminar de ordenar el segmento, p_i espera en la barrera cíclica. Una vez todos los núcleos han alcanzado la barrera, se realiza una reducción en la cual cada p_i par integra su segmento ordenado con el de p_{i+1} si existe, sobre el arreglo auxiliar, este proceso se repite intercambiando el arreglo auxiliar como el origen, y el origen como auxiliar en cada paso, hasta llegar al final de la reducción, donde el último arreglo utilizado como auxiliar contiene la totalidad del arreglo ordenado, como puede observarse en la figura 1.

En el algoritmo ***Bubble Sort***, se implementó un bucle principal que va desde el índice del último elemento (*n-1*), decrementando en 1 hasta llegar a 0, en cada ciclo se ejecuta un bucle interior que va desde cero hasta el valor actual del bucle principal. De esta manera en cada ciclo principal el elemento de mayor magnitud es llevado a la última posición denotada por el valor del índice del bucle principal. La implementación presentada fue basada en la optimización de una bandera que comienza en falso antes de cada bucle interior, cuando ocurre un intercambio en el bucle interior a la bandera se le asigna un valor verdadero, si al finalizar el bucle interior la bandera es verdadera se procede al siguiente ciclo del bucle superior, en

cambio si la bandera es falsa, todos los elementos están ordenados y se termina la ejecución [4, p. 2].

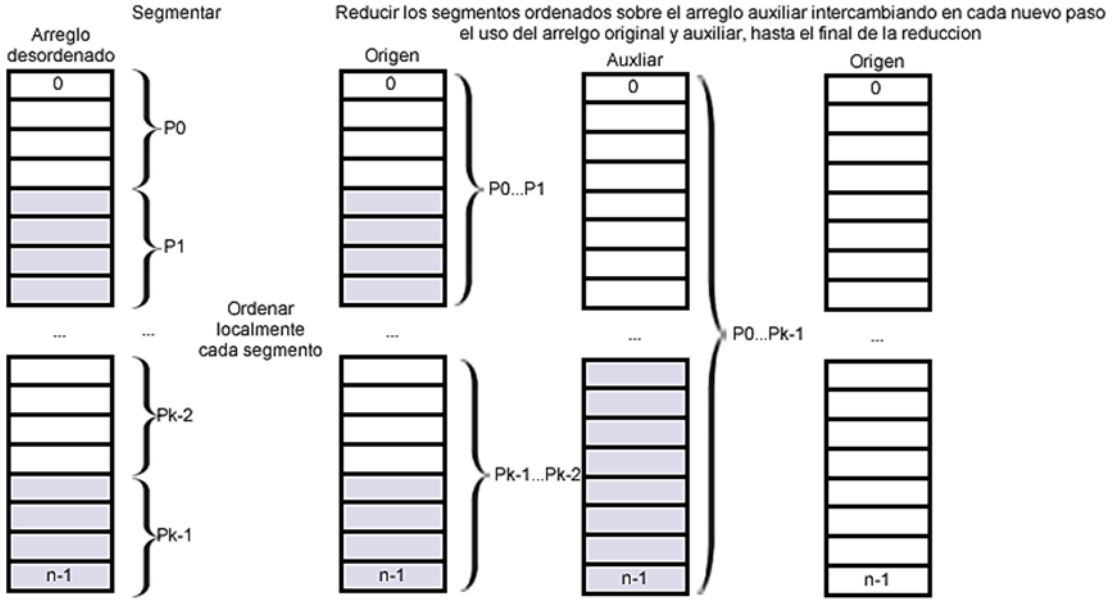

Figura 1 Proceso paralelo, ordenando por segmentos locales, y reduciendo al final.

Para la versión paralela se utilizó la forma de segmentar descrita con anterioridad en la figura 1, aplicando ***Bubble Sort*** a cada segmento de manera individual y paralela, una vez ordenados, se unen los segmentos a un solo arreglo.

El algoritmo ***Odd Even Sort*** es una mejora sobre Bubble Sort, para ejecutarse en múltiples núcleos de forma paralela. Este algoritmo opera en dos fases, la fase par y la impar. En la fase par, todos los elementos n_i pares son comparados contra n_{i+1}, en la fase impar los elementos n_i impares con comparados contra n_{i+1}, durante estas dos fases se tiene una bandera para indicar si se han intercambiado elementos, esta bandera es evaluada al finalizar las dos fases, si la bandera es verdadera se realizado un intercambio, por lo que se ejecutan nuevamente ambas fases, si la bandera es falsa, no se han realizado intercambios, por lo que el arreglo se encuentra ordenado y se finaliza la ejecución.

En la implementación paralela se utilizó un arreglo auxiliar y una barrera cíclica, a cada procesador p_i se le asigna un índice inicial acorde a la fase par o impar y el índice del procesador, se incrementa en la cantidad de procesadores en cada

iteración hasta concluir la fase donde esperan a la barrera cíclica, si esta es la primera fase se procede a la segunda, en cambio sí es la segunda fase, se valida el estado de las banderas de intercambio, si al menos una es verdadera se procede desde la primera fase, si todas son falsas, el arreglo está entonces ordenado y se finaliza la ejecución. Con este modelo se evita la modificación concurrente de los elementos en el arreglo ya que el elemento contiguo sobre el que se compara sólo es modificado o leído por un núcleo a la vez.

El algoritmo ***Rank Sort*** toma un elemento a la vez del cual hace un conteo de la cantidad de elementos menores, una vez calculado, este elemento es colocado en un arreglo auxiliar en la posición denotada por el cálculo de números menores, cuando todos los elementos han sido clasificados y colocados en el arreglo auxiliar, se finaliza la ejecución.

Para la versión paralela se utilizó la forma de segmentar descrita con anterioridad en la figura 1, aplicando ***Rank Sort*** a cada segmento de manera individual y paralela, una vez ordenados, se unen los segmentos a un solo arreglo.

El algoritmo ***Counting Sort***, ordena arreglos que contiene valores dentro de un rango [1, *m*], mediante la generación del histograma del arreglo, en el que se realiza una suma prefija, con la cual se calcula la posición de cada elemento del arreglo original, dentro de un nuevo arreglo donde son colocados en orden.

En la implementación paralela cada procesador realiza el histograma sobre un segmento del arreglo, este histograma es único y local para cada procesador, al finalizar, se reduce el histograma de todos los procesadores a uno solo, sobre el cual se realiza la suma prefija y se genera el arreglo ordenado de la misma manera que en la implementación serial.

El algoritmo ***Bitonic Sort***, se basa en secuencias bitónicas de las cuales la primera mitad se ordena ascendente y la segunda descendente de manera recursiva en segmentos que son en cada recursión la mitad del segmento original, una vez ordenadas se mezclan de regreso para obtener el arreglo ordenado, este algoritmo está diseñado para arreglos una cantidad de elementos que es una potencia de dos, en este artículo se utiliza la implementación de Hans Werner Lang que permite utilizar arreglos que no son potencias de 2 [5].

Para la versión paralela se utilizó la forma de segmentar descrita con anterioridad en la figura 1, aplicando ***Bitonic Sort*** a cada segmento de manera individual y paralela, una vez ordenados, se unen los segmentos a un solo arreglo.

En algoritmo ***Quick Sort*** al igual que en el caso de Bubble Sort, primero se divide el arreglo en dos subarreglos tomando un elemento como pivote, se itera sobre el arreglo de tal manera que los elementos menores que el pivote queden del lado izquierdo y los mayores del lado derecho, a continuación, aplica recursivamente *Quick Sort* a la parte izquierda, después nuevamente aplica *Quick Sort* a la derecha, cuando la serie de llamadas recursivas se completa, el arreglo se encuentra ordenado. Para la versión paralela se utilizó la forma de segmentar descrita con anterioridad en la figura 1, aplicando ***Quick Sort*** a cada segmento de manera individual y paralela, una vez ordenados, se unen los segmentos a un solo arreglo.

El algoritmo ***Radix Sort*** ordena los elementos por segmentos definidos por un número de bits, de derecha a izquierda en la mayoría de las implementaciones, la cantidad de bits que se usan para ordenar por segmento son estáticas. Para este comparativo se utilizó ARL (*Adaptive Left Radix*), este algoritmo es un Radix Sort de izquierda, con el número de bits determinado dinámicamente para cada segmento, cabe destacar que es un ordenamiento que no es estable [6, p. 3].

Para el comparativo paralelo se utilizó PARL (*Parallel Adaptive Left Radix*)) [7], en este algoritmo se utiliza una barrera cíclica, a cada procesador se le asigna un segmento sobre el que se calcula primero el máximo local, una vez determinada la espera de todos en la barrera cíclica, se determina el máximo local, mediante este máximo cada procesador realiza el ordenamiento ARL secuencial del primer grupo de bits obtenido del máximo sobre su segmento, una vez calculado esperan en la barrera, al alcanzarla se colocan los elementos del arreglo en tantos cubos como procesadores se estén utilizando. A continuación, cada procesador ordena a través de ARL serial el cubo asignado, desde el siguiente grupo de bits del que se usó en el primero ordenamiento, al terminar y alcanzar la barrera se copian todos los elementos en orden.

El algoritmo ***Merge Sort*** divide el arreglo en dos, y cada uno de estos nuevamente de manera recursiva hasta que el arreglo contiene un elemento, un arreglo con un

elemento se considera ordenado, en este punto ambos arreglos se encuentran ordenados por lo que se unen de manera ordenada, después de las dos llamadas de división en cada recursión, al finalizar el arreglo final está ordenado.

Para la versión paralela se utilizó la forma de segmentar descrita con anterioridad en la figura 1, aplicando ***Merge Sort*** a cada segmento de manera individual y paralela, una vez ordenados, se unen los segmentos a un solo arreglo.

El caso de prueba es un arreglo de 100000 enteros aleatorios en el rango de [1,1000]. Se realizaron mediciones con 1, 2, 3 y 4 procesadores para cada uno de los algoritmos descritos anteriormente. Las implementaciones fueron realizadas en Java 8 en el equipo descrito en tabla 1.

Tabla 1 Especificaciones del sistema donde se realizaron las pruebas.

Tipo de máquina	X86_64	Número de núcleos	4 CPUs
Sistema operativo	Windows10	Velocidad de CPU	4800 MHz
Versión de sistema	10.0.17763	Memoria Total	32635 Mbytes
JVM	1.8.0_144	Espacio Swap Total	No Swap

Fuente: elaboración propia.

Se utilizó el mismo arreglo desordenado de entrada 1000 veces para cada algoritmo y número de procesadores, este arreglo desordenado es el mismo que el utilizado en el artículo *Comparative Analysis of Sorting Methods using OpenMP*.

Adicionalmente para los algoritmos *Radix Sort* y *Counting Sort*, se realizaron pruebas con arreglos de 100000 elementos, así como 1, 2, 3, 4 5, 6, 7, 8 y 9 millones. Para el caso de Odd Even Sort se recolectó información adicional del tiempo en espera de sincronización.

En la configuración de la JVM (*Java Virtual Machine*) se incrementó el tamaño del heap y stack, para evitar el desbordamiento en los algoritmos recursivos, así como reducir las posibles llamadas al recolector de basura **GC** (GC, por sus siglas en inglés) de la instancia de la máquina virtual, también se deshabilitó la optimización a instrucciones SIMD (*Single Instruction, Multiple Data*), con la finalidad de evitar que la instancia optimice ciclos y afecte al tiempo de ejecución de algunos de los algoritmos. Los argumentos de la máquina virtual quedan de la siguiente manera:

-Xms4096m -Xmx12000m -Xss1024m -XX:-UseSuperWord

Para cada uno de los algoritmos se utilizó la media del tiempo de ejecución de las 1000 corridas usando 1, 2, 3 y 4 núcleos respectivamente.

3. Resultados

El caso de prueba es un arreglo de 100000 enteros aleatorios en el rango de [1,1000]. En la figura 2, se puede observar el tiempo de ejecución de los 8 algoritmos, realizando la variación en la cantidad de núcleos disponibles.

En el caso de *Odd Even Sort* la sincronización que se realiza al final de cada iteración limita la escalabilidad, debido a que se vuelve más costoso que el procesamiento de los números, como se muestra en la figura 3.

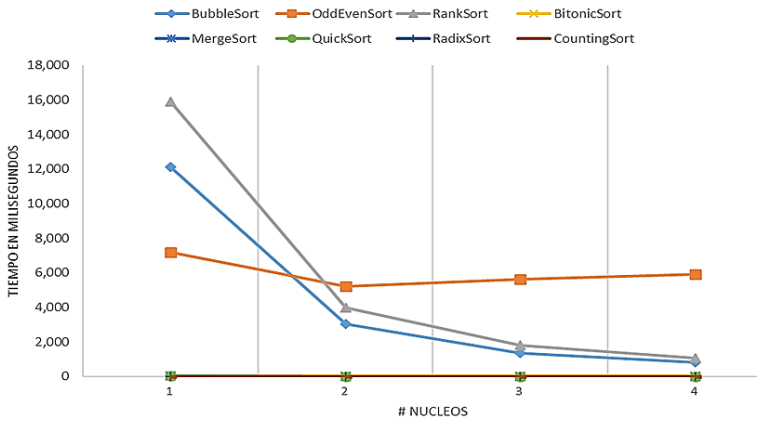

Figura 2 Tiempos de ejecución para los 8 algoritmos descritos, en 1 hasta 4 núcleos.

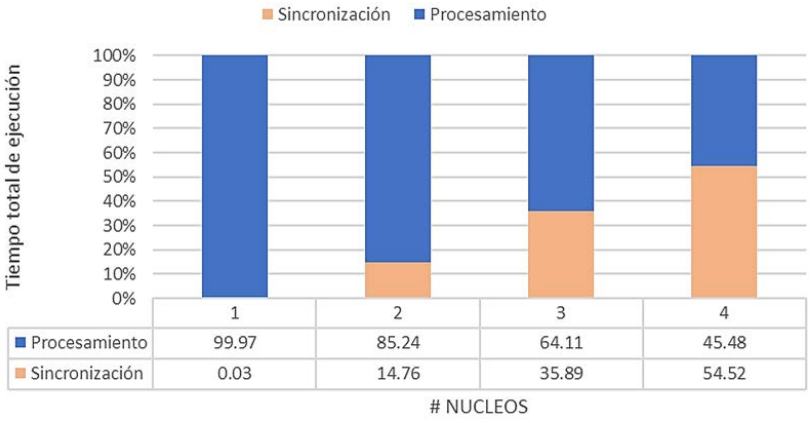

Figura 3 Tiempo usado en procesamiento y sincronización en *Odd Even Sort*.

En la figura 4, se observan los algoritmos de complejidad O(*n*Log(*n*)) y O(*n*), en el caso de *Bitonic Sort*, *Merge Sort* y *Quick Sort* se puede observar una mejora al incrementar los núcleos, sin embargo, en el caso de *Radix Sort* y *Counting Sort* es necesario observar por separado los resultados, para este tamaño de muestra incrementar los núcleos utilizados no implica una mejora en su rendimiento.

En el caso de los algoritmos *Radix Sort* y *Counting Sort*, se realizaron pruebas con arreglos de 100000 elementos, así como 1, 2, 3, 4, 5, 6, 7, 8 y 9 millones, se observa en la figura 5 para *Radix Sort* y figura 6 para *Counting Sort*, los resultados.

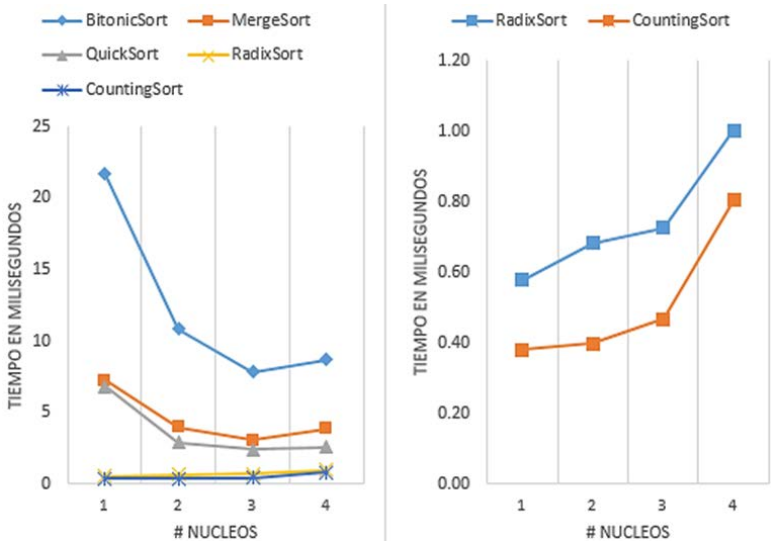

Figura 4 Tiempos de ejecución para algoritmos de menor tiempo, en 1 hasta 4 núcleos.

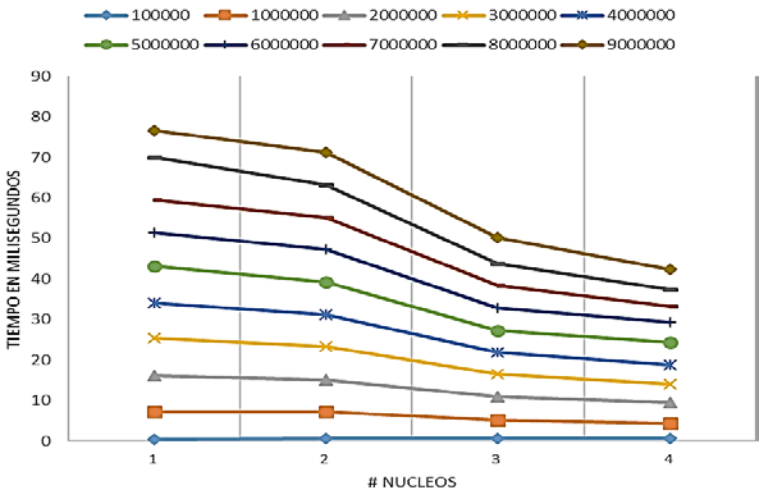

Figura 5 Tiempo ejecución *Radix Sort* 1 a 4 núcleos, diferentes tamaños de muestra.

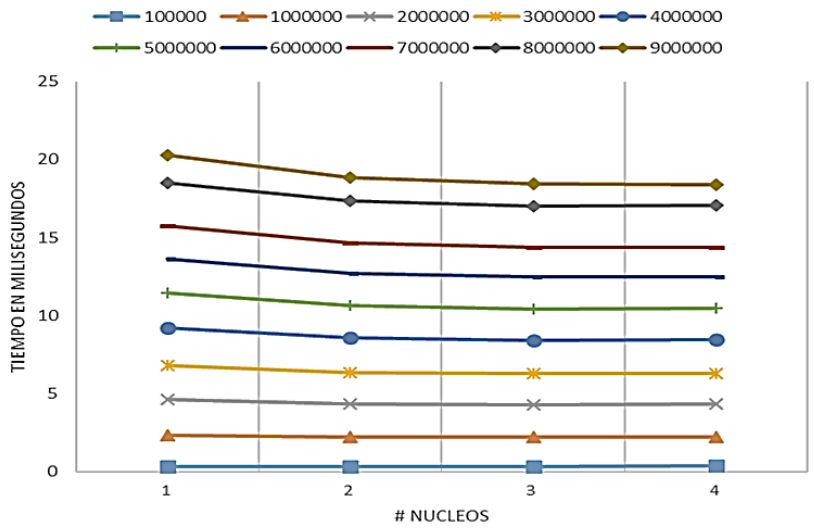

Figura 6 Tiempo ejecución *Counting Sort* 1 a 4 núcleos, diferentes tamaños de muestra.

4. Conclusiones

En el caso de *Odd Even Sort*, la sincronización al finalizar cada fase de iteración, aunque el tiempo neto de procesamiento para ordenar escala con los núcleos, se observa en la figura 4. que la sincronización al esperar en la barrera cíclica, aumenta considerablemente junto con los núcleos el tiempo total de ordenamiento.

En cambio, *Bubble Sort* y *Rank Sort*, mejoran su tiempo de ejecución, debido a que la cantidad de n/p elementos sobre la que se opera en cada segmento al ser algoritmos de complejidad O(n^2), el tiempo que tarda por cada segmento es menor que operar sobre la totalidad de los elementos.

En los casos de *Bitonic Sort*, *Merge Sort* y *Quick Sort*, el tiempo de ejecución es muy bajo, puede notarse que se obtiene un beneficio de múltiples procesadores, sin embargo, al acercarse al tiempo mínimo necesario en la JVM para lanzar los hilos, el escalado comienza a no obtener ventajas.

En el caso de *Radix Sort* y *Counting Sort*, ambos de complejidad lineal O(n), el proceso adicional de crear hilos y unir los resultados, produce que un incremento en la cantidad de procesadores, no precisamente mejore su rendimiento en los arreglos con pocos datos, pero se puede observar en las figuras 5 y 6, al incrementar la cantidad de elementos en el arreglo, el tiempo de ejecución se ve beneficiado por el incremento en núcleos de procesamiento.

5. Bibliografía y Referencias

[1] Venners, B. (1998). Inside the Java Virtual Machine. New York: McGraw-Hill.

[2] Barney, B. (n.d.). Introduction to Parallel Computing (2018): https://computing.llnl.gov/tutorials/parallel_comp/.

[3] Guerrero, E. A., Palafox, A., Serrano, J. P. & Alonzo, J. L. (2013). Comparative Analysis of Sorting Methods using OpenMP. In Supercomputing in México A Navigation Through Science and Technology (1st ed., pp. 129–139). Guadalajara: University of Guadalajara.

[4] Astrachan, O. (2003). Bubble Sort: An Archaeological Algorithmic Analysis. SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education).

[5] Lang, H. W. (2018). Bitonic sorting network for n not a power of 2 (diciembre 2018): www.iti.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/oddn.htm.

[6] Maus, A. (2003). ARL, a faster in-place, cache friendly sorting algorithm.

[7] Maus, A. (2011). A full parallel radix sorting algorithm for multicore processors.