Tarea 02 – Programación vectorial

Felipe Leviñir Astudillo, felipe.levinir@alumnos.uv.cl

Resumen: En la actualidad existen diversos métodos y criterios para ordenar conjuntos de datos. Esencialmente, existen dos tipos de algoritmos de ordenamiento: los que dependen de comparaciones y cuyas complejidades se encuentran entre lo polinomial y lo logarítmico, y los de conteo, que asumen ciertas cosas sobre el conjunto de datos y cuyos tiempos son lineales. En este estudio se realizó una medición temporal al ordenamiento de diversos conjuntos de datos numéricos, mediante métodos de programación vectorial con las intrínsecas que ofrece Intel para realizar vectorización y se comparó el desempeño temporal del ordenamiento logrado con la función std::sort() que ofrece C++ para ordenar conjuntos de datos. Para realizar el ordenamiento vectorial se realizaron una serie de pasos. Inicialmente se utilizó una Sorting Network comparando los valores máximos y mínimos en las columnas de matrices cuadradas (4x4), luego fue necesario trasponer dicha matriz para obtener un conjunto semiordenado de datos. Posteriormente se tomaron secuencias de ocho elementos ordenados de los pasos anteriores y se ordenaron con el método Bitonic Sorter y finalmente se utilizó un Bitonic Merge Network para finalizar el ordenamiento vectorial. Se espera que el ordenamiento vectorial tenga un mejor desempeño a medida que aumenta el volumen de datos.

1. Introducción

Los algoritmos forman parte del día a día de las personas. Algunos ejemplos claros de algoritmos pueden ser las instrucciones de una receta de cocina, las indicaciones de un medicamento o una rutina del día a día. Un algoritmo se puede ver como una secuencia de acciones que se deben llevar a cabo para obtener una solución de un determinado problema finito. Según Edsger Dijkstra[1], un algoritmo se corresponde con una descripción de un patrón de comportamiento, expresado en términos de un conjunto finito de acciones.

El ordenamiento es una de las actividades realizadas con más frecuencia en el procesamiento de datos. Uno de los objetivos principales de este proceso es aligerar la posterior búsqueda de elementos dentro de un conjunto de datos, al utilizar algoritmos de búsqueda que requieren un conjunto de datos previamente ordenados. Otra de las finalidades de obtener un conjunto de datos ordenado es la de emitir salidas ordenadas por algún campo en particular, para lo cual se requiere también el proceso previo de ordenamiento. Estas, entre otras, son razones poderosas para mejorar continuamente los procesos de ordenamiento de datos buscando su máxima eficiencia.

Existen en la práctica diferentes maneras de ordenar un conjunto de datos [2], desde el ordenamiento BubbleSort hasta complejos procedimientos como el Algoritmo HeapSort, pasando por el ordenamiento por Inserción, el Quicksort, el ShellSort, el MergeSort, entre otros. Todos ellos tienen la misma finalidad, ordenar el conjunto de datos, pero difieren en la forma como operan sobre ellos para conseguir el objetivo. Además, desde el punto de vista algorítmico, la complejidad de cada algoritmo es distinta y su eficiencia depende en muchos casos del conjunto de datos a ordenar.

Para este estudio se realizó un ordenamiento de datos mediante técnicas de programación vectorial con las intrínsecas que ofrece Intel [3]. Las funciones intrínsecas se utilizan a menudo para implementar explícitamente la vectorización y la paralelización en lenguajes que no abordan tales construcciones. Algunas interfaces de programación de aplicaciones (API), por ejemplo, AltiVec y OpenMP, utilizan funciones intrínsecas para declarar, respectivamente, operaciones vectorizables y con reconocimiento de multiprocesamiento durante la compilación. El compilador analiza las funciones intrínsecas y las convierte en matemáticas vectoriales o código objeto de multiprocesamiento apropiado para la plataforma de destino. Algunos elementos intrínsecos se utilizan para proporcionar restricciones adicionales al optimizador, como valores que una variable no puede asumir.

El objetivo principal del estudio es realizar una comparación temporal de la función std::sort()[4] que ofrece C++ para realizar ordenamiento de datos numéricos en memoria, versus las técnicas de programación vectorial con las

intrínsecas que ofrece Intel para realizar un ordenamiento vectorizado haciendo uso del procesador para realizar los cálculos de ordenamiento. Para las mediciones de tiempo se estudió como la función sort de C++ ordeno un conjunto de n datos desordenados. Para el ordenamiento vectorial se ordenará parcialmente un conjunto de n datos desordenados llamado $conjunto_1$, obteniendo un conjunto de datos parcialmente ordenado llamado $conjunto_2$, este $conjunto_2$ para que este totalmente ordenado, se le aplicará la función sort también, pero se espera que como este segundo conjunto este parcialmente ordenado, el tiempo de ordenamiento sea menor a diferencia de ordenar el conjunto de datos totalmente desordenado al parcialmente ordenado.

Se espera que a medida que vaya aumentando el tamaño del problema, la solución con métodos de programación vectorizada sea más eficiente que la función std::sort(), dado que es menor el acceso a memoria que realiza en comparación a esta última.

2. Materiales y Métodos

Para desarrollar este trabajo se deben tener en cuenta que el algoritmo trabajado solamente hará uso de CPU para realizar sus cómputos. Se utilizó un servidor con un CPU Intel(R) Xeon(R) Platinum 8260 a 2.40 GHz con una arquitectura x86_64 [5], la conexión al servidor fue realizada mediante un cliente SSH (Como por ejemplo PuTTy) [6] como se muestra en la figura 1.

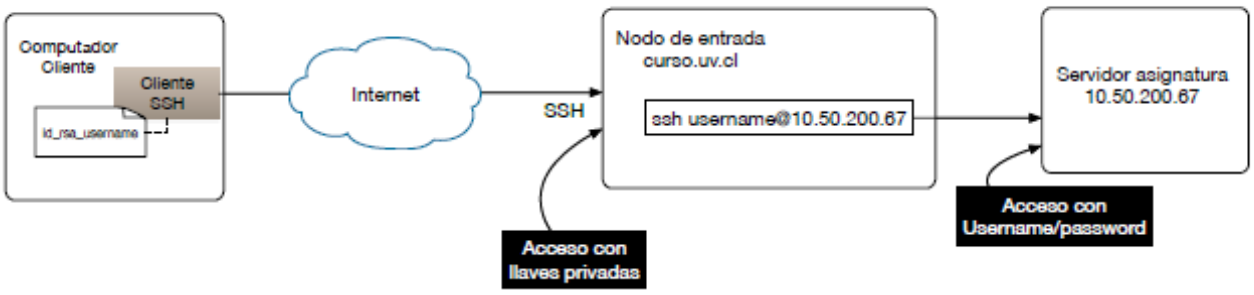


Figura 1. Diagrama de conexión con el servidor

Una vez conectado en el servidor, se necesita un compilador de C++ (lenguaje de programación en el cual está implementado el algoritmo), que debe poseer el servidor. En caso de no contar con un compilador de C++, se debe instalar uno. Así mismo, para poder trabajar con el código, se necesita un editor de texto, la mayoría de estos vienen incorporados en la consola de trabajo, en el experimento se utilizó el editor de texto VI [7], pero se puede utilizar cualquier otro editor de texto que posea la terminal, como por ejemplo NANO[8].

2.1. Secciones del algoritmo.

El algoritmo trabajado [9] consta de diversas secciones en las cuales se van realizando variados cálculos de ordenamiento y también la lectura de los datos a analizar. Para efectos prácticos del trabajo se dividió el algoritmo en nueve secciones con el fin de explicar su funcionalidad y el diseño de estas. El diseño del algoritmo vectorizado consta de cuatro pasos fundamentales para crear el ordenamiento vectorial de los datos, explicados en la sección 2.2 de este documento. Las otras secciones del algoritmo se explican en la sección 2.3 del documento.

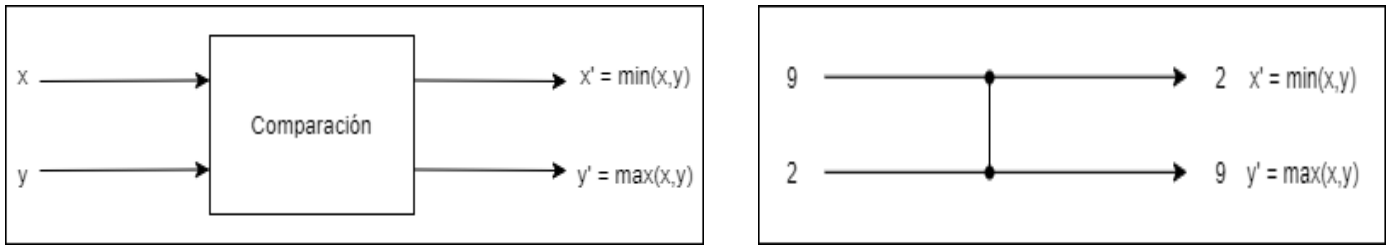
2.2. Secciones y diseño del algoritmo vectorizado.

Para las primeras cuatro secciones del algoritmo se crearon distintas funciones que fueron la implementación del diseño del diseño que se explica a continuación.

2.2.1. Sorting Network

Inicialmente se necesita crear una "Sorting Network" [10]. Esta se compone únicamente de cables y comparadores. Un comparador es un dispositivo con dos entradas, x e y, y dos salidas, x' e y', que realiza la siguiente función de comparación como se muestra en la figura 2.a.

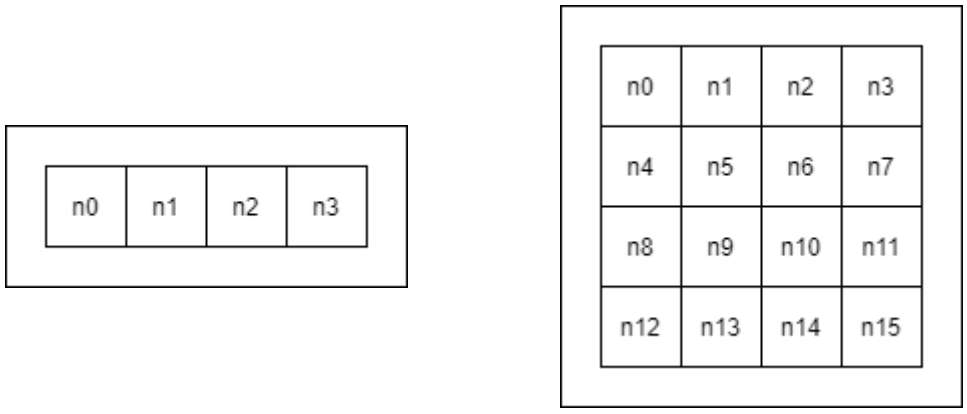
Hay una representación gráfica conveniente de la "Sorting Network" como se puede ver en la figura 2.b. Una línea horizontal representa cada entrada de la red de clasificación y una conexión entre dos líneas representa cada comparador que compara los dos elementos y los intercambia si el de la línea superior es mayor que el de la línea inferior. La entrada de la red de clasificación está a la izquierda de la representación. Los elementos en la salida se ordenan y el elemento más grande migra a la línea inferior.



(**a**) (**b**)

Figura 1. Funcionamiento del Sorting Network. (**a**) Teoría básica del ordenamiento (**b**) Ejemplo práctico del ordenamiento.

Para el trabajo realizado y en la primera sección del algortimo se trabajaron registros de ciento veintiocho bits, los cuales son vectores que pueden almacenar cuatro valores como se ven en la figura 3.a. Para poder aplicar el Sorting Network se necesita por lo menos otro vector para tener dos filas y realizar las comparaciones que se requieren para ordenar los valores, pero para efectos prácticos del trabajo y para lograr una eficiencia a la hora de ordenar los valores se trabajó con cuatro vectores que almacenan cuatro valores numéricos dentro de ellos como se puede ver en la figura 3.b.



(**a**) (**b**)

Figura 3. Explicación gráfica de cómo se trabajarán los datos. (**a**) Vector básico (**b**) Matriz creada a partir de cuatro vectores.

En el siguiente ejemplo de la figura 4 se puede ver como se comparan los valores de los registros mediante el Sorting Network, ordenando las columnas de la matriz mostrada en la figura 3.b. Para esto se realizaron cinco pasos que se evidencian en el código en la primera sección de trabajo. En el primer paso se compara el mínimo valor del registro 0

con el máximo valor del registro 2. Luego en el segundo paso se compara el mínimo valor del registro 1, con el máximo valor del registro 3. En el tercer paso se compara el el mínimo valor del registro 2 con el máximo valor del registro 3. En el cuatro paso se compara el mínimo valor del registro 0 con el máximo valor del registro 1 y finalmente en el quinto paso se compara el mínimo valor del registro 1 con el máximo valor del registro 2. Con esto obtenemos un matriz con los valores de su columnas ordenados de menor a mayor.

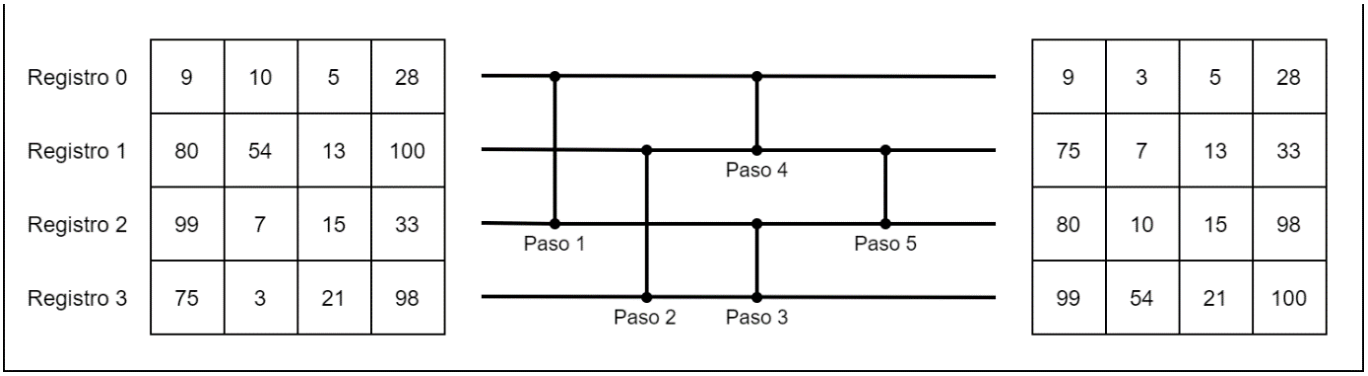


Figura 4. Ejemplo de Sorting Network.

2.2.2. Matriz traspuesta

Para efectos del ordenamiento de n cantidad de datos tenemos bloques de diesciseis valores parcialmente ordenados, ya que, al tener solamente las columnas de las matrices de cuatro por cuatro ordenadas, tenemos conjuntos parcialmente ordenados, pero nosotros necesitamos tener las filas de la matriz ordenada, ya que, esa es la forma en que van llegando los vectores a la memoria, para ordenar nuestros conjuntos de diesciseis números parcialmente ordenados, tenemos que trasponer la matriz que obtenemos al realizar el Sorting Network como se ve en la figura 5. Lo cual se realiza en la segunda sección del código. Con esto ahora tenemos las filas de la matriz ordenadas y nuevamente obtenemos un conjunto parcialmente ordenado de datos.

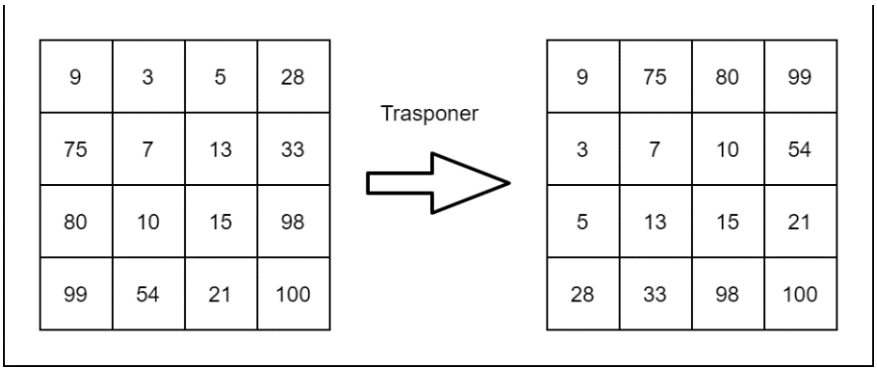


Figura 5. Ejemplo de una matriz traspuesta.

2.2.3. Bitonic Sorter

Para solucionar este desorden de datos de nuestro conjunto parcialmente ordenado, tenemos que avanzar a la tercera sección del código, en la cual se implementó otro método de ordenamiento, el cual es aplicar el Bitonic Sorter [11], dónde este es capaz de ordenar ocho número pre ordenados, trabajando con dos vectores de cuatro elementos parcialmente ordenados, el primer vector debe estar ordenado de menor a mayor, el cual ya lo tenemos de los pasos anteriores, pero el segundo vector debe estar ordenado de mayor a menor, por lo que el primer paso dentro del Bitonic Sorter es cambiar

el orden del segundo vector que recibe, ya que, este está recibiendo vectores ordenados de menor a mayor de los pasos anteriores. En la figura 6 se puede ver como el primer paso del Bitonic Sorter.

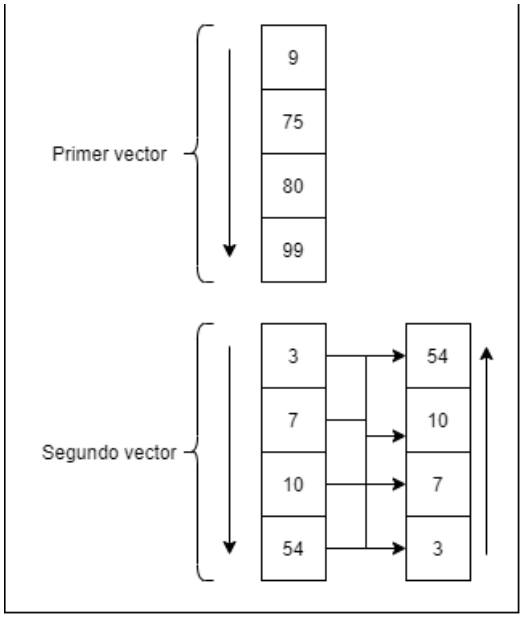


Figura 6. Primer paso del Bitonic Sorter

Luego de reordernar el segundo vector se comienzan a realizar una serie de comparaciones entre pares con valores mínimos y máximos como se ve en la figura 6. El algoritmo divide el vector en dos subvectores y estos sucesivamente entre dos para obtener otros subvectores. Luego compara los elementos entre pares (subvectores), el primer elemento de la primera mitad con el primer elemento de la segunda mitad y así sucesivamente. En el supuesto de encontrar algún elemento de la segunda mitad más pequeño que de la primera, estos elementos se intercambian, siempre comparando el menor elemento con algún elemento mayor y dejando el menor en la parte superior y el elemento mayor en la parte inferior.

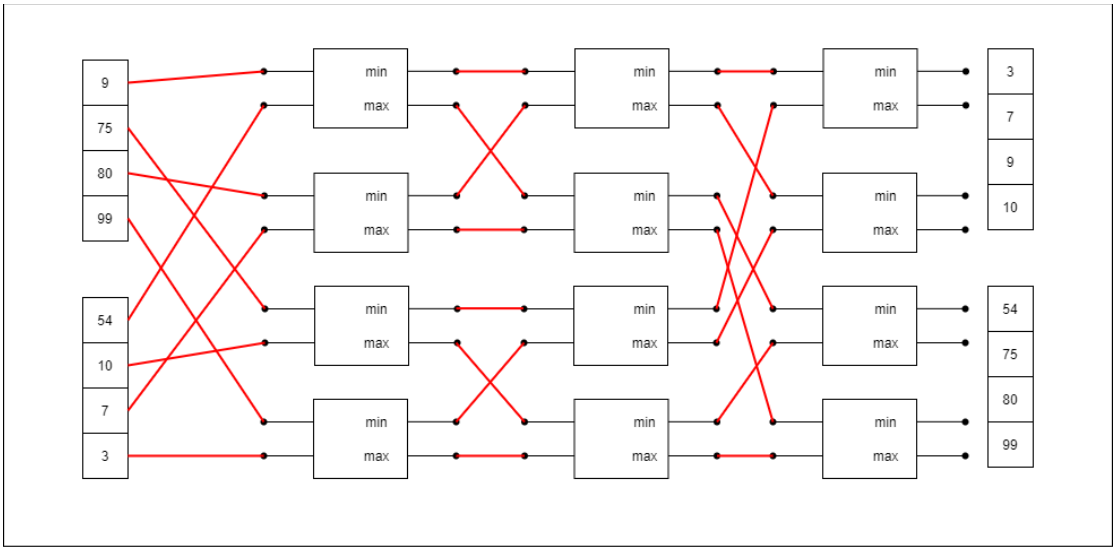


Figura 7. Aplicación del Bitonic Sorter

Ahora con la implementación del Bitonic Sorter tenemos ocho elementos ordenados de menor a mayor, pero estarían faltando ordenar los otro ocho elementos de la matriz que habíamos creado en pasos anteriores. Por lo tanto, avanzamos a la sección cuatro del algoritmo, la cual consiste en generar una Bitonic Merge Network, que ordenará los dieciséis elementos de la matriz que generamos anteriormente.

2.2.4. Bitonic Merge Network

Bitonic Merge Network [12] es la red que puede unir dos secuencias de entrada ordenadas en una secuencia de salida ordenada. Adaptamos Bitonic Sorter para crear la red de fusión Merge. La red de fusión se basa en el siguiente supuesto: Dadas dos secuencias ordenadas, si invertimos el orden de la segunda secuencia y luego conectamos las dos secuencias, la secuencia resultante es bitónica. El Bitonic Merge Network recibe por parámetros secuencias de dieciséis elementos ordenados a partir de dos secuencias bitónicas de entrada de orden ocho, como se muestra en la figura 8.

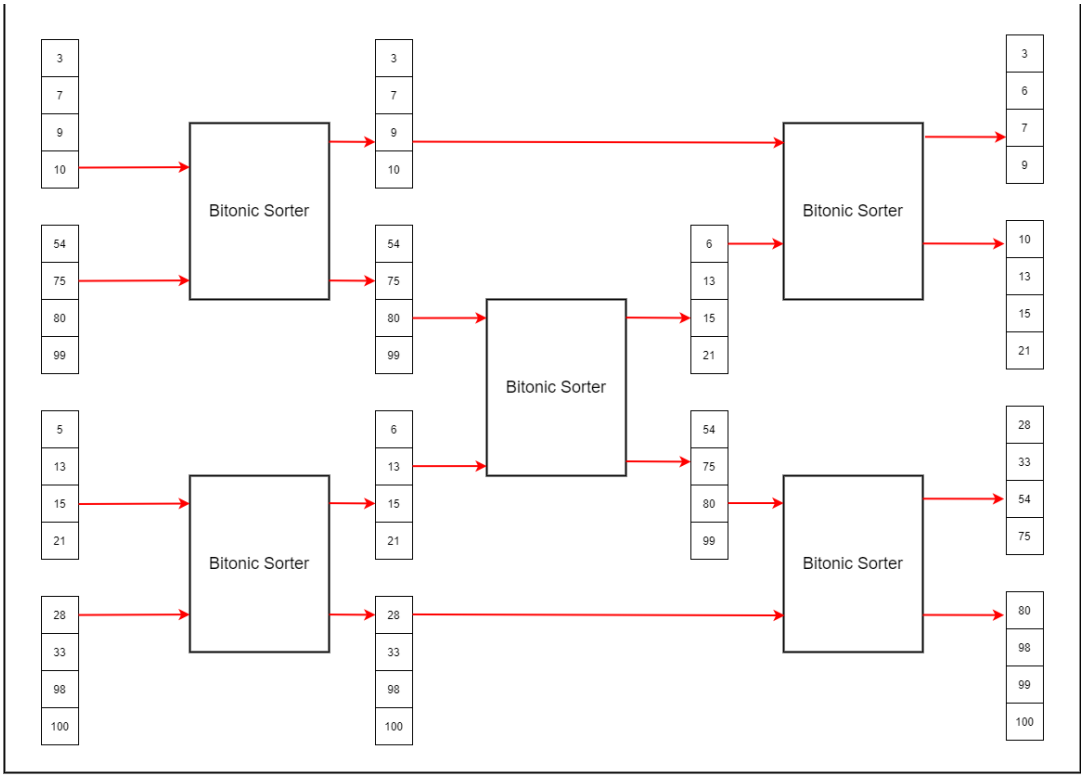


Figura 8. Aplicación del Bitonic Merge Network

Finalmente, con las primeras cuatro secciones del código podemos ordenar secuencias de dieciséis elementos y obtenemos un conjunto parcialmente de datos ordenados, con esto finalizaría el ordenamiento vectorial y el nuevo arreglo de datos parcialmente ordenado se ordenará con la función sort para que el conjunto total de datos este totalmente ordenado.

2.3. Secciones restantes del algoritmo.

En las demás secciones del algoritmo se realizan pequeñas operaciones, en la quinta sección en particular se define una función para indicar la forma de utilizar el código ejecutable y el resto de las secciones se declaran variables, se carga a memoria los datos a analizar y se realizan las mediciones de tiempo correspondientes para cada caso. En la octava sección en particular se llaman a las funciones de ordenamiento vectorial para realizar el trabajo de ordenamiento al archivo que se carga a memoria. Todas las secciones restantes se explican a detalle a continuación en las subsubsecciones de este documento. En esta quinta sección como se mencionaba anteriormente se crea una función que indica la forma de ejecutar el c digo una vez compilado mediante el Makefile del código.

2.3.1. Función de uso.

En esta quinta sección como se mencionaba anteriormente se crea una función que indica la forma de ejecutar el código una vez compilado mediante el Makefile.

2.3.2. Comienzo de la función principal.

En esta sexta sección del código comienza la función principal del algoritmo, la función main(), en donde esta todo el cuerpo funcional del código, en la cual se llaman a las funciones de las primeras cuatro secciones (funciones encargadas de vectorizar los datos explicadas en la sección 2.2 de este documento). En esta sección dentro de la función main() tenemos la declaración de las variables para realizar las mediciones temporales, contamos de cinco variables temporales para realizar diversos análisis, además hacemos uso de la función de la sección cinco para realizar la lectura del archivo y almacenar la variable en memoria.

2.3.3. Carga de archivos a memoria y ordenamiento con sort.

En la séptima sección del código se realizaron tres operaciones relevantes, la primera de ellas es la carga a memoria del archivo a trabajar al cual se le mide el tiempo en que se demora en cargar a memoria, esta primera carga a memoria se almacenará en la variable llamada $m1$, por lo que ahora en memoria tenemos un archivo de n elementos llamado $m1$ el cual debemos ordenado solamente con la función sort(). A continuación, se ordena el archivo $m1$ con la función sort() y se hace la toma de tiempo correspondiente para ver cuánto se tarda en ordenar el archivo completo. Finalmente se realiza una segunda carga a memoria llamada $m2$, la cual es el archivo para trabajar de forma vectorial.

2.3.4. Ordenamiento vectorial.

En la octava sección del algoritmo es donde se llaman a las funciones para realizar el ordenamiento de datos con técnicas de vectorización, se define un arreglo el cuál ira recibiendo cuatro datos que serán los vectores para trabajar, luego mediante un ciclo de repetición for haremos que se realice la vectorización de los datos para un tamaño de problema n e iremos cargando a memoria nuestro conjunto de datos parcialmente ordenado. Luego nuestro conjunto de datos parcialmente ordenado es sometido a la función sort para completar el ordenamiento de los datos. Cabe mencionar que se realizó la medición temporal para el ordenamiento vectorial y el ordenamiento de los datos con la función sort.

2.3.5. Muestra de datos temporales.

Finalmente, en la última sección del código, se muestra por pantalla los resultado del tiempo de las mediciones temporales del ordenamiento vectorial y el ordenamiento sort que ordena el conjunto parcialmente ordenado y se muestra el SpeedUp que alcanza el código.

3. Pruebas y Resultados.

Las pruebas del algoritmo consistieron en tomar cinco archivos de distintos tamaños $10^3, 10^4, 10^5, 10^6, 10^7$, a los cuales se les medió el tiempo que tardaban en ejecutarse. Para hacer las pruebas de tiempo, se colocaron seis distintos timers, para tomar el tiempo de ejecución en las distintas secciones del código que se querían evaluar. Los cuales son:

1. El primer timer esta para tomar el tiempo de la primera vez que se cargan a memoria los datos.
2. El segundo timer toma el tiempo de ejecución de la función sort con los datos desordenados.
3. El tercer timer toma el tiempo el de la segunda vez que se carga a memoria los datos.
4. El cuarto timer mide el tiempo de ordenamiento vectorial.
5. El quinto timer toma el tiempo de la función sort con los datos parcialmente ordenados.
6. El sexto y ultimo timer esta dispuesto en todo el código main() para realizar la toma de tiempo del SpeedUp.

Para el análisis temporal de ejecución del programa se realizaron tres pruebas con el fin de tener más datos y llegar a analizar si varía el tiempo de ejecución en algún momento determinado.

En la tabla 1, 2 y 3 se pueden evidenciar los tiempos que se tardo el algoritmo en realizar las distintas operaciones que se le solicitaban realizar (Los cinco primeros timers). Los resultados que entrega el experimento es el tiempo que tardo el algoritmo en ejecutarse, tiempo que se está midiendo en ms. Cada tabla tiene una figura asociada para ayudar a la visualización de los datos de forma gráfica.

3.1. Tablas.

Tabla 1. Primera medición de temporal de ejecución.

Tamaño del problema	Ordenamiento en memoria con sort()	Ordenamiento luego de vectorizar	Ordenamiento vectorizado	Carga a memoria m1	Carga a memoria m2
10^3	0	0	0	0	0
10^4	1	1	0	0	0
10^5	17	16	2	8	8
10^6	206	199	28	85	82
10^7	2362	2275	273	1111	964

Tabla 2. Segunda medición de temporal de ejecución.

Tamaño del problema	Ordenamiento en memoria con sort()	Ordenamiento luego de vectorizar	Ordenamiento vectorizado	Carga a memoria m1	Carga a memoria m2
10^3	0	0	0	0	0
10^4	1	1	0	0	0
10^5	17	16	2	8	8
10^6	198	195	26	93	88
10^7	2303	2250	270	1100	958

Tabla 3. Tercera medición de temporal de ejecución.

Tamaño del problema	Ordenamiento en memoria con sort()	Ordenamiento luego de vectorizar	Ordenamiento vectorizado	Carga a memoria m1	Carga a memoria m2
10^3	0	0	0	0	0
10^4	1	1	0	0	0
10^5	16	16	0	7	6
10^6	200	195	27	91	86
10^7	2318	2256	271	1106	974

3.2. Gráficos.

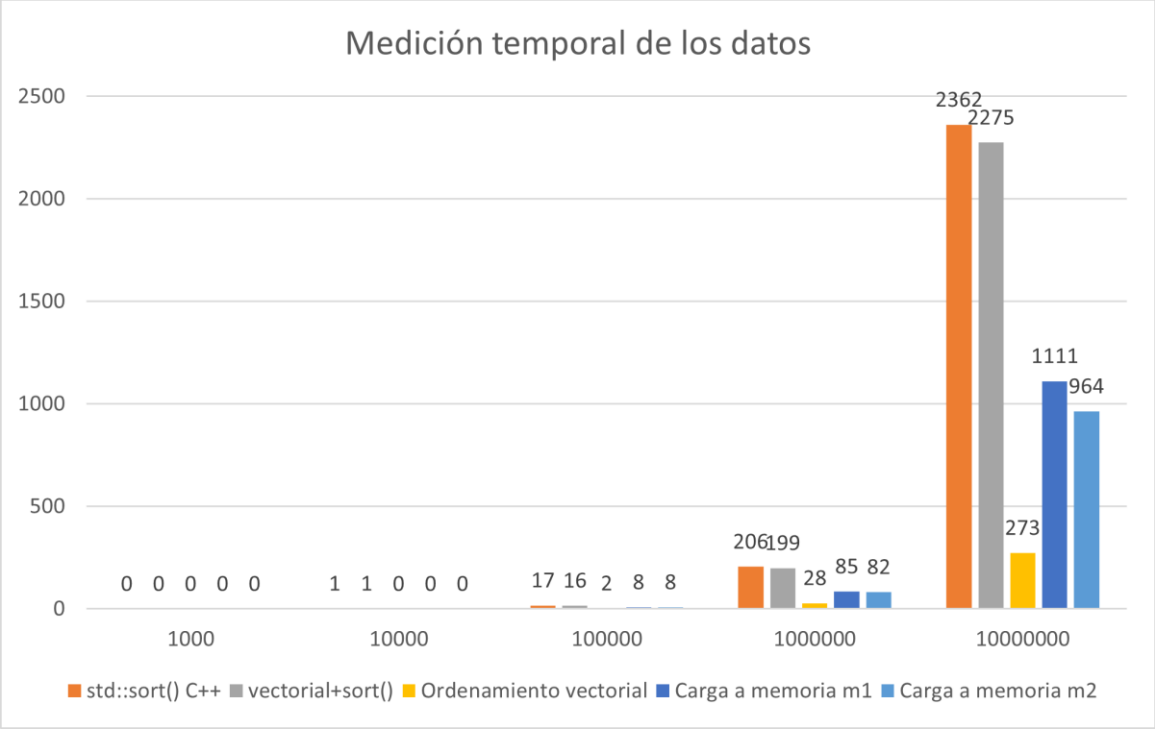


Figura 9. Grafico de la tabla 1

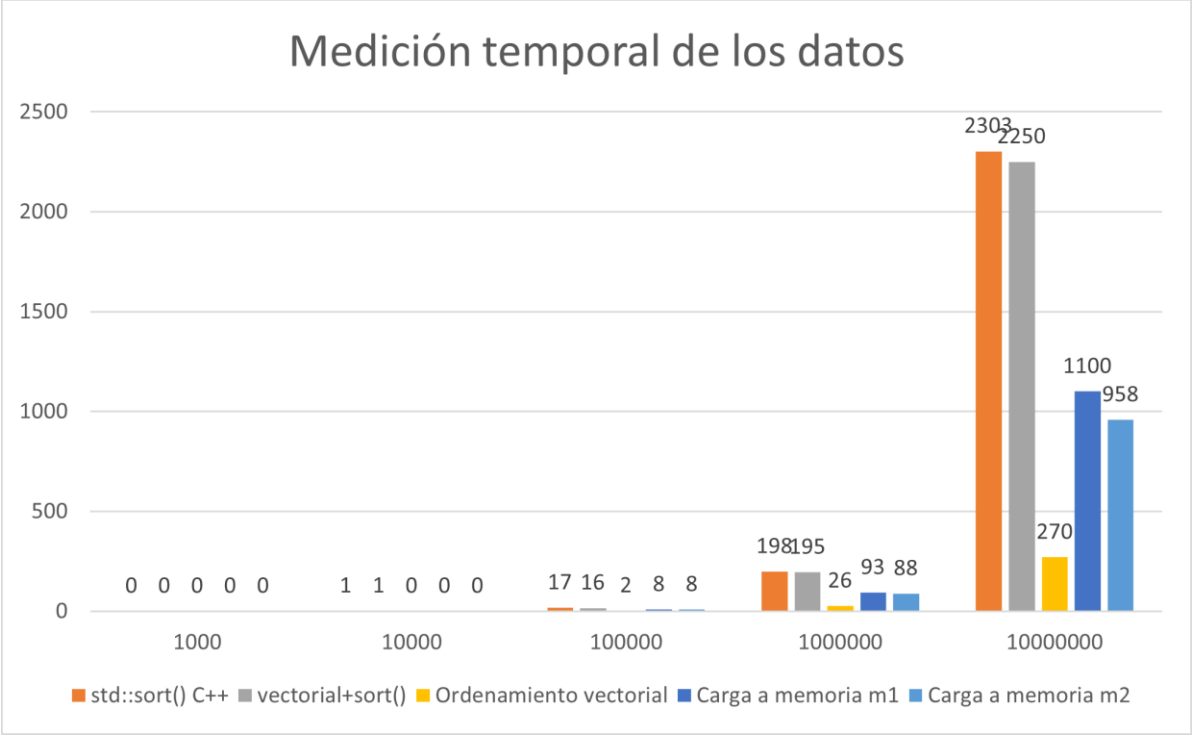


Figura 10. Grafico de la tabla 2

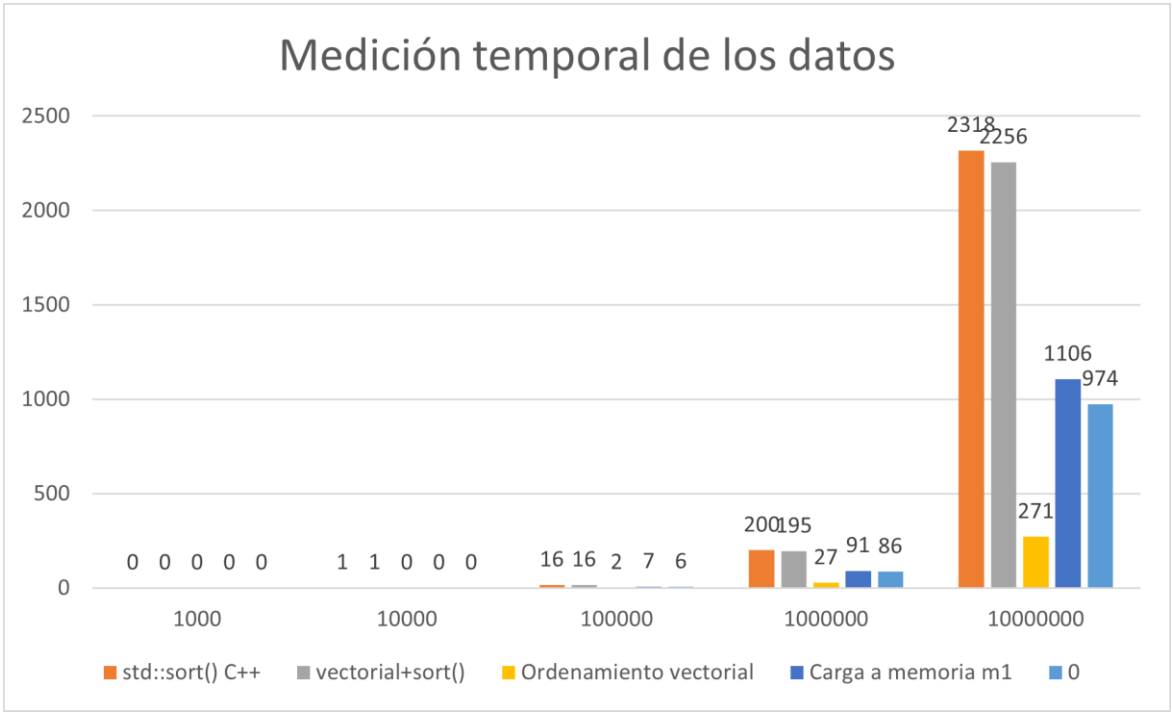


Figura 11. Grafico de la tabla 3.

3.3. Análisis de los resultados.

Se puede notar en la experimentación que para los tamaños de problema $10^3 y\ 10^4$ los tiempos de ejecución para ambas formas son prácticamente cero, es imperceptible el tiempo de ejecución que estas realizan, recordando que los valores en las tablas 1, 2 y 3 están dispuestos en milisegundos. Para el experimento realizado con 10^5 datos se puede ver que el método de ordenamiento sort no se ve afectado al ordenar un archivo desordenado en su totalidad versus un archivo parcialmente ordenado. Además, podemos comenzar a ver que el tiempo de ordenamiento vectorial aún sigue siendo cero.

Podemos notar una mejora de tiempo en los experimentos realizado con $10^6 y\ 10^7$ datos, aunque la mejora de tiempo con 10^6 es mínima , es en los 10^7 donde podemos comenzar a ver una leve mejora de tiempo a la hora de realizar el ordenamiento de los datos, llegando a ganar unos cuantos milisegundos a la hora de tener que ordenar datos parcialmente ordenados.

Analizando los tiempos de ejecución del ordenamiento vectorial se puede deducir que el crecimiento temporal es directamente proporcional a la cantidad de datos que se deben de ordenar, ya que, se puede observar que a medida que se aumenta una potencia de 10, el crecimiento temporal aumenta en un potencia de 10, esto se puede notar a partir del experimento con 10^5 datos y se espera que el crecimiento de los datos se vea como en la figura 12. Por lo que se cree que el crecimiento temporal es exponencial.

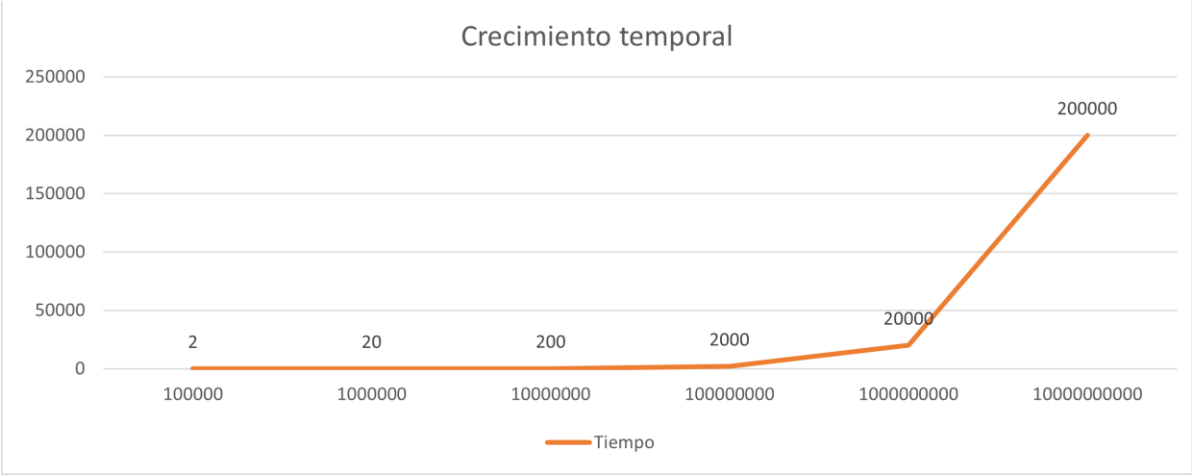


Figura 12. Proyección estimada temporal del odenamiento vectorial.

Además de realizar todo el análisis temporal de los ordenamientos, se realizó un medición de tiempo al código en general para ver el SpeedUp secuencial que el algoritmo tiene a la hora de analizar las distintas cantidades de datos.

En los siguientes gráficos se podrá apreciar los SpeedUp alcanzado para los tres experimentos que se realizaron.

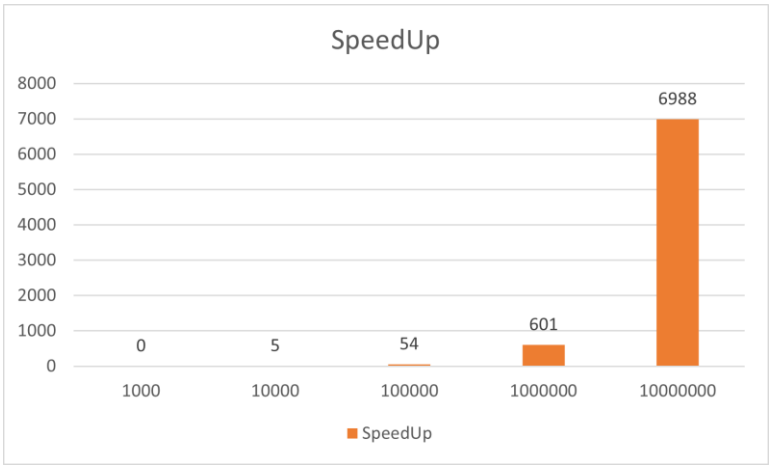


Figura 13. Primer SpeedUp

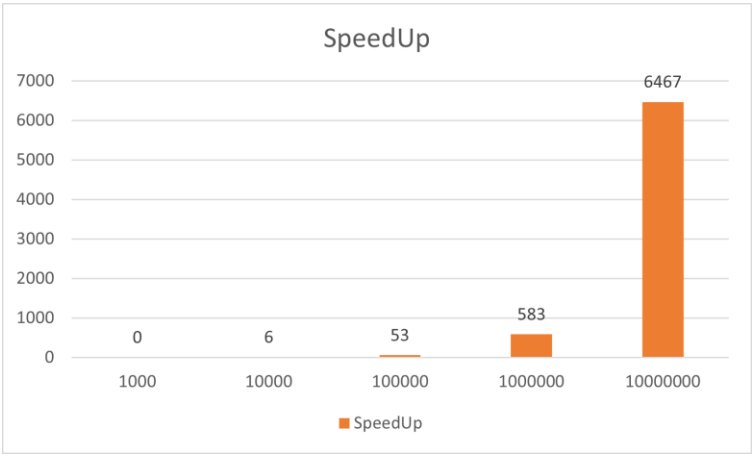


Figura 14. Segundo SpeedUp

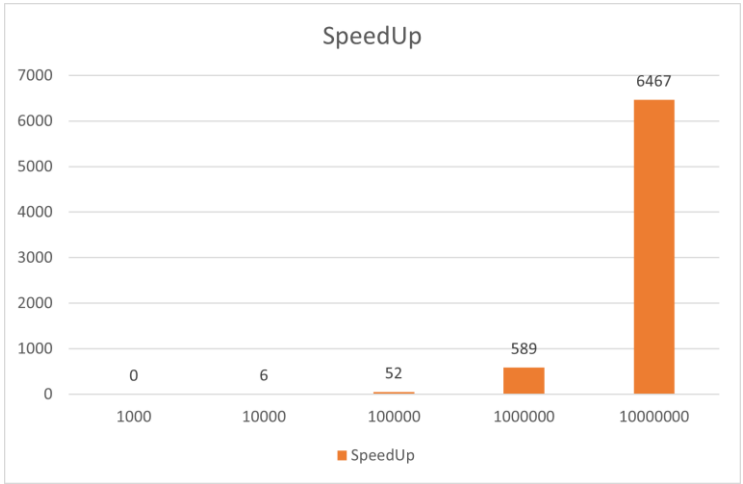


Figura 15. Tercer SpeedUp

4. Discusión y conclusiones

Luego de observar y analizar los casos planteados, podemos notar que a medida que se aumenta el tamaño del problema aumenta el tiempo de ejecución al momento de tener que ordenar archivos de tamaño 10^n, pero es en la cantidad de 10^5 datos en donde se comienza a ver una pequeña medición de tiempo más notable, ya que, en las pruebas con $10^3 y\ 10^4$ los tiempos que se obtuvieron fueron prácticamente cero al momento de ordenar los datos de manera normal versus el ordenamiento vectorial, por lo que para el análisis general tomaremos en cuenta los valores a partir desde los 10^5 datos, que es como se mencionaba anteriormente, donde se puede comenzar a realizar un análisis de los tiempos de ejecución.

En el conjunto de datos de 10^5 se comienza a ver que el ordenamiento vectorial es bastante más eficiente que el ordenamiento tradicional, ya que, el tiempo que tardo en ordenar la función sort el archivo completo fue de diecisiete milisegundos en los dos primeros casos y de dieciséis milisegundos en la última prueba, y el tiempo de ordenamiento vectorial fue tan solo de dos milisegundos en las tres pruebas, por lo que podemos decir que al tener que realizar menos acciones en memoria, el ordenamiento vectorial fue mucho más veloz, pero este solo deja valores parcialmente ordenados. Luego el tiempo de ordenamiento que tuvo que realizar la función sort fue un poco más veloz en comparación a la primera vez, ya que, al tener que ordenar un conjunto de datos parcialmente ordenados, ganó tan solo un milisegundo de tiempo con respecto al ordenamiento normal en dos pruebas y cero en la última. Por ende, en este experimento, pese a que se ganó mucho tiempo en el ordenamiento vectorial al no tener que ingresar tantas veces a la memoria principal, no podemos decir que hubo una ganancia significativa a la hora de tener que ordenar los datos, ya que, los tiempos de ejecución son muy parecidos y no hay una enorme diferencia en las mediciones temporales.

Son en las pruebas con $10^6 y\ 10^7$ en donde podemos comenzar a ver el aumento en los tiempos de ejecución en comparación a las pruebas anteriores. Con el tamaño de $10^6 y\ 10^7$ podemos observar de los gráficos y las tablas que el tiempo de ordenamiento con la función sort (trabajando con los datos totalmente desordenados) varía entre los ciento noventa y ocho hasta los doscientos seis milisegundos, y entre los dos mil trecientos tres hasta los dos mil trecientos sesenta y dos milisegundos, según las pruebas realizadas en cada caso respectivamente. Luego tenemos que el ordenamiento de datos de manera vectorial varía entre los veintiséis a veintiocho milisegundos y entre los doscientos setenta hasta los doscientos setenta y tres milisegundos para el estudio con $10^6 y\ 10^7$ respectivamente en las tres pruebas.

Por lo que de igual manera que en el tamaño de 10^5 se puede apreciar que el tiempo de ejecución vectorial para $10^6\ y\ 10^7$ es notablemente más bajo que al trabajar con la función sort (al tener que ordenar datos totalmente desordenados), pero de igual manera el conjunto de datos que obtenemos es un grupo de datos parcialmente ordenados, el cual se debió de ordenar con la función sort para tener un conjunto de datos totalmente ordenado. Por lo que podemos analizar que, pese a que se gana bastante tiempo de ejecución al realizar operaciones vectoriales, el tiempo que tarda en ordenar los conjuntos de datos parcialmente ordenados a un grupo totalmente ordenado no se gana tanto tiempo en comparación al no realizar ordenamiento vectorial, esto se puede deber a los accesos de memoria que se realizaron en el código, ya que, las dos manera de cargar los archivos fueron similares, quizás para el ordenamiento vectorial abría que aplicar otro tipo de técnica de carga a memoria para reducir el tiempo.

Si de alguna manera, se pudiera reducir significativamente el tiempo de ordenamiento del conjunto parcialmente ordenado al momento de tener que dejarlo totalmente ordenado, se podría hablar de una mejora considerable, ya que, los tiempos de ordenamiento vectorial son bajos con respecto a la función sort y se espera que el crecimiento temporal sea directamente proporcional a la cantidad de datos que se deben de ordenar, pero esto debe de ser comprobado en otro experimento, pero si esto se llegará a cumplir se podría tomar una correcta decisión a la hora de tener que implementar algún algoritmo de ordenamiento, ya que, se tendría de antemano una noción del tiempo que se demoraría en ejecutar algún ordenamiento de datos.

5. Referencias

1. DE, N., 2021. Edsger Wybe Dijkstra – Historia de la Informática. [online] Histinf.blogs.upv.es. Available at: <https://histinf.blogs.upv.es/2010/10/28/dijkstra/> [Accessed 22 April 2021].
2. Es.wikipedia.org. 2021. *Algoritmo de ordenamiento - Wikipedia, la enciclopedia libre*. [online] Available at: <https://es.wikipedia.org/wiki/Algoritmo_de_ordenamiento#:~:text=En%20computación%20y%20matemáticas%20un,la%20relación%20de%20orden%20dada.> [Accessed 1 June 2021].
3. Software.intel.com. 2021. *Intel® Intrinsics Guide*. [online] Available at: <https://software.intel.com/sites/landingpage/IntrinsicsGuide/> [Accessed 1 June 2021].
4. Educative: Interactive Courses for Software Developers. 2021. *What is the std::sort() function in C++?*. [online] Available at: <https://www.educative.io/edpresso/what-is-the-stdsort-function-in-cpp> [Accessed 1 June 2021].
5. Openaccess.uoc.edu. 2021. [online] Available at: <http://openaccess.uoc.edu/webapps/o2/bitstream/10609/79526/8/Estructura%20de%20computadores_Módulo%206_Programación%20en%20ensamblador%20%28x86-64%29.pdf> [Accessed 23 April 2021].
6. Astudillo, G., 2021. Acceso servidor asignatura Programación Paralela.
7. Unirioja.es. 2021. EDITOR VI. [online] Available at: <https://www.unirioja.es/cu/enriquez/docencia/Quimica/vi.pdf> [Accessed 23 April 2021].
8. Carbonell, L., 2021. nano, un editor de texto para la terminal. Personalización y productividad. - Atareao. [online] Atareao. Available at: <https://atareao.es/software/programacion/nano-un-editor-de-texto-para-la-terminal/> [Accessed 23 April 2021].
9. GitHub. 2021. *FelipeLevinir/Tarea02*. [online] Available at: <https://github.com/FelipeLevinir/Tarea02> [Accessed 1 June 2021].
10. Es.wikipedia.org. 2021. *Red de ordenamiento - Wikipedia, la enciclopedia libre*. [online] Available at: <https://es.wikipedia.org/wiki/Red_de_ordenamiento> [Accessed 1 June 2021].
11. GeeksforGeeks. 2021. *Bitonic Sort - GeeksforGeeks*. [online] Available at: <https://www.geeksforgeeks.org/bitonic-sort/> [Accessed 1 June 2021].
12. Pages.cs.wisc.edu. 2021. *Sorting on hypercubic networks*. [online] Available at: <http://pages.cs.wisc.edu/~tvrdik/17/html/Section17.html> [Accessed 1 June 2021].