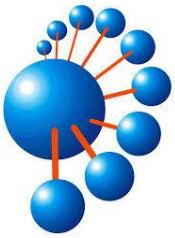
Universidad de Concepción
Facultad de Ingeniería



Medición sistemática del consumo energético de algoritmos de ordenamiento y búsqueda

Integrante:	Leonardo De La Fuente
Profesor Guía:	José Fuentes
Colaborador:	Zheng Li
Comisión:	Diego Seco
	Roberto Asín

Junio de 2021, Concepción.

Resumen Ejecutivo

En esta memoria se realiza un estudio sobre algoritmos de búsqueda y ordenamiento para establecer la relación que existe entre el tiempo de ejecución del algoritmo y la eficiencia energética que posee.

Para ello se escogieron tres algoritmos de ordenamiento: Merge Sort, Heap Sort y Counting Sort, en conjunto con dos de búsqueda: Binary Search y Binary Search Tree. Cada algoritmo será ejecutado bajo ciertas configuraciones predefinidas de inputs que fueron previamente elaboradas. Además, la recolección de datos será llevada mediante la herramienta *Perf,* la que permitirá almacenar los valores obtenidos de la experimentación. Cabe añadir que cada implementación se evaluará en tres dispositivos, siendo dos computadores portátiles y el restante un contenedor que se almacenará en una de ellas.

A partir de los valores obtenidos de la ejecución de cada uno de los algoritmos con las distintas configuraciones y dispositivos, se elaborará un análisis enfocado en el tiempo de ejecución y consumo energético. Esto es acompañado por una serie de métricas que podrían detallar o ayudar a discernir el comportamiento del algoritmo con respecto a las principales métricas de tiempo y energía.

Los principales resultados de los análisis indicaron que el tiempo de ejecución no siempre es directamente proporcional al consumo energético, es decir, la ejecución más rápida pueda que no sea la que menos energía requiera. Otro punto adicional es el hecho de que un algoritmo puede tener distintas tendencias en base al input con el cual se está ejecutando. En el caso de los algoritmos que poseen una implementación recursiva e iterativa, esta última presenta mejores resultados con respecto a la eficiencia energética y de tiempo que su contraparte recursiva. Por otro lado, lo que involucra al contenedor, bajo ciertos inputs y algoritmos genera mayor eficiencia que un dispositivo con sistema operativo.

Contenido

Capítulo 1. Introducción

En un principio, el desarrollo de software tiene como objetivo primordial cumplir con las solicitudes y objetivos de los clientes o usuarios, junto a una optimización de desempeño, la que usualmente implica utilizar las estructuras de datos y algoritmos más adecuados. Esto conlleva a tener programas que se miden en base a los tiempos de respuesta y ejecución que estos poseen, lo cual, en principio, es el punto más importante para los clientes o usuarios.

En la actualidad, se han generado grandes cambios, no sólo en lo que incumbe al ámbito de la informática y ciencias de la computación, sino desde un punto de vista ambiental y ecológico que ha repercutido en todo el mundo. Dado lo anterior, es que se debería considerar otro aspecto al momento de desarrollar software, el cual es la eficiencia energética que este posee.

1.1 Descripción del problema

Un tema que en el mundo de la programación en general no suele ser considerado al momento de desarrollar software, es el consumo energético que este genera, siendo eventualmente un apartado desconocido para gran parte del personal involucrado [1]. A su vez, no deja de ser un punto de gran relevancia, puesto que existen ejemplos de aquello. El primero se relaciona con *Edge Computing*, el cual consiste en realizar el procesamiento de datos de manera local en dispositivos cuyos recursos de memoria y batería son limitados. Esto permite evitar en la medida posible enviar información a Data Centers o Clouds. Sin embargo, esto implica una mayor carga de trabajo, lo que se traduce en una elevación del consumo energético del dispositivo, afectando el rendimiento de equipos que dependen de baterías [2]. Otro ejemplo de mayor envergadura es la huella de carbono que puede producir un algoritmo de *Artificial Intelligence*, más específicamente *Deep Learning*, para el cual su ejecución puede ser equivalente a la contaminación producida por 5 automóviles en un año, incluyendo su fabricación [3]. Para ambos casos, un concepto y conocimiento adecuado del consumo energético de algoritmos, permitiría escoger las estructuras de datos más aptas para aquellas implementaciones.

Su bajo nivel de apreciación puede deberse en principio a la estereotipada pero razonable hipótesis de que el algoritmo que menor tiempo de ejecución posea tendrá una mayor eficiencia energética. En otras palabras, el tiempo de ejecución y el consumo

energético de un algoritmo son directamente proporcionales. Esta idea planea ser puesta a prueba para dos categorías de algoritmos: ordenamiento y búsqueda.

1.2 Objetivo general

El principal objetivo que se busca es medir la eficiencia energética que presentan algunos algoritmos de ordenamiento y búsqueda, con el fin de determinar si es posible generar cambios en la actual forma de desarrollar software.

1.3 Objetivos específicos

- Obtener el consumo energético de los núcleos, paquete y RAM, junto con el tiempo de ejecución para cada uno de los algoritmos.
- Recabar variadas métricas relacionados a la ejecución de los algoritmos para ayudar a comprender su comportamiento, tales como cantidad de instrucciones, ciclos de CPU, fallas de caché, entre otras.
- Evaluar la ejecución de los algoritmos bajo diversos inputs.
- Determinar la correlación entre tiempo de ejecución y consumo energético de los algoritmos estudiados.
- Realizar una comparativa de la ejecución que tienen los algoritmos en un sistema operativo nativo y un contenedor.
- Publicar todo el material recolectado en repositorios de libre acceso para dar la posibilidad a la comunidad y cualquier persona interesada en conocer los resultados obtenidos.

1.4 Alcances y Limitaciones

Los algoritmos utilizados representan una versión en particular de estos, por ende, existe la posibilidad de que variantes obtengan resultados diferentes a los mostrados en este trabajo. Además, siguiendo esta misma línea, al reproducir la experimentación, pueden diferir los resultados obtenidos al utilizar una máquina con distintas características. Sin embargo, las proporciones y tendencias de las principales métricas de tiempo y energía deberían mantenerse en base al algoritmo y hardware utilizado para su ejecución.

Con respecto al input, existe una gran variedad de combinaciones posibles de elementos para los distintos tamaños. Para evitar sesgos en su elección, se generaron diversos inputs de manera semi – aleatoria, limitando la cota inferior y superior de los posibles valores de cada elemento. De esta manera, la probabilidad de estar presente en casos extremos disminuye.

Por otro lado, la herramienta, en este caso *Perf* presenta limitaciones generadas por la máquina en la cual se desarrolla la experimentación, dado que existen métricas que no le es posible obtener. También, se presentan situaciones en las cuales la herramienta se ve incapacitada de capturar resultados, esto se debe a la precisión y requerimientos mínimos que necesita para llevar a cabo lo solicitado. Otro punto es que sus mediciones obtienen valores en base a todo proceso que se esté llevando a cabo en el momento de la ejecución del algoritmo, por lo que se establecieron las mismas condiciones. Con respecto a las limitaciones presentes producto de la máquina, se optó por capturar métricas que se encontraran disponibles en los dos equipos de trabajo, teniendo un total de 17 métricas.

1.5 Observación

Cabe señalar que en principio la propuesta de memoria consideraba medir el impacto de batch processing, el cual es el proceso mediante el cual un computador completa lotes de trabajo, a menudo simultáneamente, en orden secuencial e ininterrumpido. Sin embargo, esto fue modificado y reemplazado por el análisis de un contenedor, debido a la gran influencia que tiene actualmente sobre cambiar la manera en la que se desarrolla software.

Capítulo 2. Marco Teórico

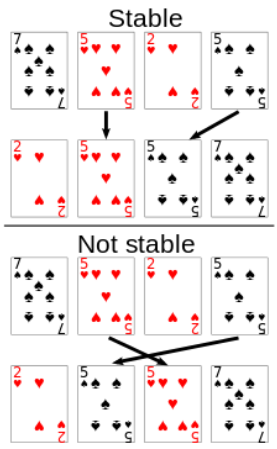
Este apartado introduce los dos tipos de algoritmos a estudiar, los cuales son de ordenamiento y búsqueda, posterior a eso, se detallará con mayor profundidad cada uno de ellos. Adicionalmente, se explicarán diferentes conceptos relacionados a la arquitectura que presenta un equipo en general, incluyendo memorias caché, paquetes y características de la misma índole. También, se detallará la herramienta *Perf,* puesto que es la que permitió llevar a cabo la recolección de datos. Además, se justificará la elección del lenguaje de programación utilizado para ejecutar los algoritmos. Por último, se relatará en que consiste un contenedor y las diferencias que presenta ante una máquina virtual.

2.1 Algoritmos de ordenamiento

Los algoritmos de ordenamiento son requeridos para organizar conjuntos de datos basándose en el valor o clave de cada elemento y el criterio estipulado para su ejecución. Un ejemplo cotidiano de aquello se puede observar cuando a partir de un grupo de estudiantes, es necesario establecer un orden en base a su apellido, el cual generalmente suele ser de forma ascendente.

En el área de la computación, el ordenamiento de datos cumple un rol de gran importancia, ya sea como labor propia o dentro de procedimientos más robustos. A su vez, se han desarrollado variadas técnicas que cumplen con este objetivo, cada una con características específicas. Esto implica tener ventajas y desventajas frente al resto, siendo dos parámetros las principales diferencias: tiempo de ejecución y memoria requerida.

Una característica en particular que poseen los algoritmos de ordenamiento es que se clasifican en dos grupos: estables y no estables. El primer caso consiste en mantener un relativo pre - orden total. Esto significa que para aquellos conjuntos de datos que presenten dos e incluso más valores o claves iguales, el resultado del ordenamiento garantiza que se respete la disposición que tenían. En cambio, los algoritmos no estables carecen de la seguridad que tendrán las posiciones de los valores o claves repetidas [4, 5].



Un ejemplo de aquello es el que se puede apreciar en la Figura 2.1, donde se desea ordenar de manera ascendente el conjunto de cartas {7,5,2,5}, dando como resultado {2,5,5,7}. Como se observa en la parte superior de la figura, el algoritmo estable retorna en segunda posición al 5 de corazones, el que se encuentra antes del 5 de picas. Por otro lado, el algoritmo no estable puede o no retornar la misma combinación, dado que como se visualiza en la parte inferior de la figura, el 5 de picas se encuentra antes que el 5 de corazones.

Figura 2.1. Algoritmos estables y no estables [39]

2.1.1 Merge Sort

La idea principal de este algoritmo es la de dividir el problema en subproblemas más pequeños, bajo el lema “divide y vencerás”. Pertenece a los algoritmos de ordenamiento estable, presentando una complejidad $O(nlog(n))$, donde n es el tamaño del conjunto [6, 7, 8, 36, 37].

La estructura básica que posee para ordenar un conjunto es la siguiente:

- Si $n \leq 1$, entonces el conjunto ya se encuentra ordenado.
- Si $n > 1$, se divide el conjunto por la mitad y se llama así mismo con cada subproblema. Su condición de término está dada por el cumplimiento de la relación $l < r$, donde l y r son la primera y última posición del actual conjunto, respectivamente, tal como se observa en el Algoritmo 1.
- Ordenar los subconjuntos mediante mezcla de tal manera que la unión sea un conjunto ordenado, como se aprecia en el Algoritmo 2.

Un ejemplo práctico es el que se puede apreciar en la Figura 2.2, la cual tiene como objetivo ordenar el arreglo compuesto por los elementos [8, 0, 3, -1, 5, 7, 2]. Dado que el tamaño del arreglo es superior a 1, se inicia el proceso de división del conjunto hasta que la primera posición es igual o mayor a la última para cada subconjunto. Posteriormente, se inicia el ordenamiento de los elementos mediante la mezcla de los subconjuntos, lo que genera el conjunto ordenado.

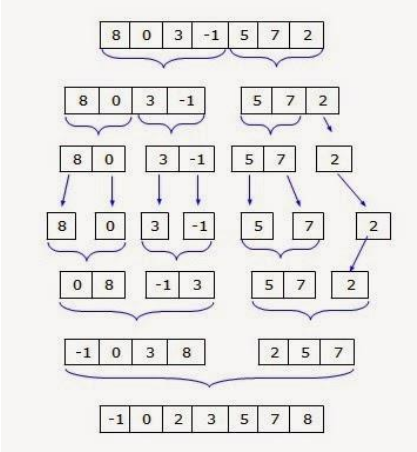


Figura 2.2. Ejemplo ilustrativo de Merge Sort. [40]

Algoritmo 1: Pseudocódigo Merge Sort

Input: Array, l, r
Output: Array ordenado
if $(l < r)$ **then**
    Mitad = $[\frac{l+r}{2}]$
    MergeSort(Array, l, mitad)
    MergeSort(Array, mitad + 1, r)
    Merge(Array, l, mitad, r)

Algoritmo 2: Pseudocódigo Merge

Input: Array, l, r, mitad
Output: Array ordenado
N1 ← mitad – l + 1
N2 ← r - mitad
Crear un arreglo para cada mitad temp1[N1] y temp2[N2]
for *i* ← 0 **to** *N1* **do**
    temp1[i] ← Array[l + i]
for *i* ← 0 **to** *N2* **do**
    temp2[i] ← Array[mitad + 1 + i]
i ← 0, j ← 0, k ← l
while (*i* < *N1* and *j* < *N2*) **do**
    if (*temp1[i]* ≤ *temp2[j]*) **then**
        Array[k] ← temp1[i]
        i ← i + 1
    else
        Array[k] ← temp2[j]
        j ← j + 1
    k ← k + 1
while (*i* < *N1*) **do**
    Array[k] ← temp1[i]
    i ← i + 1
    k ← k + 1
while (*j* < *N2*) **do**
    Array[k] ← temp2[j]
    j ← j + 1
    k ← k + 1

2.1.2 Counting Sort

Su idea se basa en contar el número de valores o claves distintas que existen dentro del conjunto de elementos. Posee ciertas restricciones, entre las más importantes se encuentra que se debe conocer tanto la cota inferior como superior del conjunto de elementos. Además, solo puede ser implementada para valores contables en un rango determinado, por ejemplo, en un intervalo de tamaño *n,* se consideraría números enteros, pero no los números reales. Cabe señalar que es ineficiente en conjuntos relativamente pequeños de tamaño, pero que contienen valores muy dispersos en relación con el menor elemento y el más grande. Pertenece a los algoritmos estables, donde dispone de una complejidad $O(n + k)$, siendo k el tamaño del arreglo auxiliar, el cual está definido por el elemento de mayor valor [9, 10, 11].

Su funcionamiento consta de la siguiente estructura, tal como se observa en el Algoritmo 3:

- Determinar el intervalo de los elementos del conjunto, es decir, cota inferior y superior.
- Crear un arreglo Count de tamaño $k + 1$ y otro Output de igual tamaño del conjunto que se quiere ordenar.
- Asignar el valor 0 para cada posición del nuevo arreglo Count.
- Tras esto, recorrer todos los elementos a ordenar y contar el número de apariciones de cada elemento e indicarlo en el arreglo Count.
- Modificar el nuevo arreglo, de modo que cada elemento de cada índice almacene la suma acumulada.
- Asignar los valores en el conjunto Output, dado las repeticiones que se almacenaron en el nuevo arreglo.
- Copiar todo el conjunto Output en el arreglo original.

Un ejemplo ilustrativo es el que se puede observar en la Figura 2.3, la cual presenta un arreglo inicial Array = [5, 7, 5, 2, 1, 1]. Como primer paso, identifica el valor más pequeño y el más grande, siendo estos el 1 y 7, respectivamente. A continuación, crea un arreglo auxiliar Count de tamaño $k + 1 = 8$, al cual se le asigna valores 0 para cada posición. También se crea otro arreglo auxiliar Output de igual tamaño que el conjunto inicial. Una vez realizado esta acción, se comienza a contar las veces que se encuentra un elemento

en el arreglo inicial, indicando esta cantidad de manera acumulada en los índices del arreglo Count. Por último, asignar los valores en el arreglo Output, dado las repeticiones de cada elemento que se encuentran en los índices del arreglo Count, para posteriormente copiar el conjunto Output en el arreglo inicial.

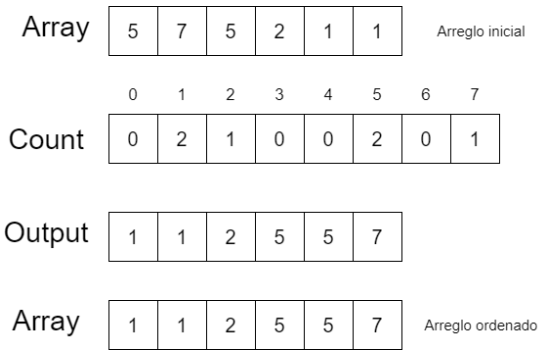


Figura 2.3. Ejemplo ilustrativo de Counting Sort. [41]

Algoritmo 3: Pseudocódigo Counting Sort

```
Input: Array, size
Output: Array ordenado
Maximo ← 0
for j ← 0 to size - 1 do
        if (Array[j] > Maximo) then
                Maximo ← Array[j]
Crear dos arreglos, uno auxiliar Output[size]
y otro en base al elemento más grande
Count[Maximo]
Inicializar cada posición del arreglo Count
con el valor 0
Contar cada valor que posee Array e
indicarlo en Count
for i ← 1 to Maximo do
        Count[i] ← Count[i] + Count[i – 1]
for i ← 0 to Array[i] do
        Output[Count[Array[i]] – 1] ←
         Array[i]
        --Count[Array[i]]
for i ← 0 to Array[i] do
        Array[i] ← Output[i]
```

2.1.3 Heap Sort

Está basado en la estructura de datos Binay Heap, la cual es un árbol binario completo que tiene como particularidad que la raíz posee el elemento más pequeño o grande, según se defina. Además, todos los nodos deben ser menor o mayor que los hijos, ya sea para ordenamiento descendente o ascendente, respectivamente. Consiste en almacenar todos los elementos del arreglo a ordenar en un montículo o heap, y luego extraer el nodo que queda como raíz del montículo en sucesivas iteraciones obteniendo el conjunto ordenado. Esto se puede observar en el Algoritmo 4, el cual es acompañado por

la función Heapify del Algoritmo 5 para realizar los intercambios de posición de los elementos. Se categoriza como algoritmo no recursivo, además de ser no estable, presentando una complejidad $O(nlog(n))$ [12, 13].

Su estructura básica para ordenamiento ascendente está compuesta por los siguientes pasos:

a) Construir un montículo de tipo Binary Heap, utilizando los elementos del arreglo, los cuales pueden ver modificadas sus posiciones originales.
b) Se intercambia en el montículo la raíz con el último nodo, dado que la raíz es parte del arreglo ya ordenado.
c) Se elimina el último nodo y se disminuye en uno el tamaño del montículo.
d) Se verifica que se cumpla la propiedad del Binary Heap. En caso contrario, se intercambian las posiciones de los nodos hasta que cada padre sea mayor que sus hijos y la raíz sea el nodo con mayor valor.
e) Repetir los pasos b), c) y d) mientras el tamaño del montículo sea mayor a uno.

Un ejemplo que permite visualizar el comportamiento de Heap Sort, es el que se observa en la Figura 2.4, la cual posee un arreglo con los elementos [4, 10, 3, 5, 1], los cuales se desean ordenar de manera ascendente. Como primer paso, crear el montículo de tipo Binary Heap, que posea el elemento más grande en la raíz y que cada padre sea mayor a sus hijos. Una vez completado este paso, se intercambia el último nodo con la raíz, y se elimina este último del montículo, además, se disminuye en uno el tamaño del montículo. Posteriormente, se verifica nuevamente que se cumpla la propiedad del Binary Heap y se repite el proceso de intercambiar el último nodo con la raíz, mientras que el tamaño del montículo sea mayor a uno.

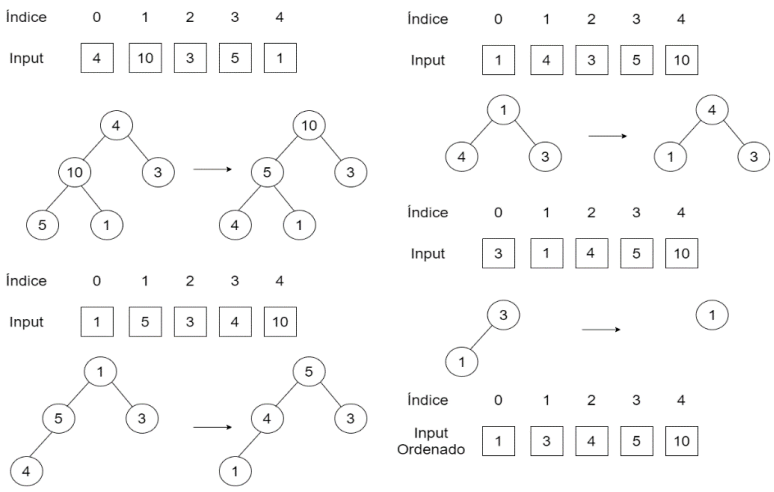


Figura 2.4. Ejemplo ilustrativo de Heap Sort. [42]

Algoritmo 4: Pseudocódigo Heap Sort

```
Input: Array, size
Output: Array ordenado
for i ← size/2 − 1  to i ≥ 0 do
        Heapify(Array, size, i)
        i ← i - 1
for i ← size - 1  to i ≥ 0 do
        Intercambiar la primera posición de
        Array con la actual
        Heapify(Array, i, 0)
        i ← i - 1
```

Algoritmo 5: Pseudocódigo Heapify

```
Input: Array, size, i
Output: Array ordenado
Largo ← i
L ← 2 * i + 1
R ← 2 * i + 2
if (L < size and Array[L] > Array[Largo])
then
        Largo ← L
if (R < size and Array[R] > Array[Largo])
then
        Largo ← R
if (Largo ! = i) then
        Intercambiar la posición actual del
        Array por la que se encuentra en
        Array[Largo]
        Heapify(Array, size, Largo)
```

2.2 Algoritmos de búsqueda

Los algoritmos de búsqueda están diseñados con el objetivo de determinar la existencia dentro de un conjunto de datos de un elemento en particular y/o la posición en la que este se encuentra. Un ejemplo de aquello y que suele ser constantemente requerido es ubicar el registro correspondiente a cierta persona en una base de datos, lo cual puede retornar lo solicitado o indicar que no existe dicha información [14].

Al igual que el tipo de algoritmo visto anteriormente, este presenta variadas técnicas desarrolladas, diferenciándose principalmente por tiempo de ejecución y memoria requerida. Sin embargo, existen casos que requieren un prerrequisito, el cual es poseer un conjunto de elementos ordenados, como es el caso de Binary Search o búsqueda binaria.

2.2.1 Binary Search

Es un algoritmo eficiente para determinar si un elemento se encuentra en un arreglo ordenado. Consiste básicamente en determinar si el valor objetivo se encuentra en la mitad del arreglo, en caso de no estar presente, se continua la búsqueda en una de las mitades

restantes, específicamente en la podría encontrarse. Aunque la idea es simple, implementar la búsqueda binaria correctamente requiere atención a algunos detalles como su condición de término y cálculo del punto medio. Presenta distintas variaciones, las cuales se pueden clasificar en: recursiva e iterativa. Además, posee una complejidad $O(log(n))$, siendo n el tamaño del arreglo [15, 16].

Su funcionamiento consta por los siguientes pasos para encontrar un elemento T, tal como se observa en el Algoritmo 6:

a) Definir dos variables; L y R, la primera con valor 0 y la segunda con valor $n - 1$.
b) Si $L \geq R$, la búsqueda termina sin retornar el elemento.
c) Se define una variable $M = [\frac{(L+R)}{2}]$, que indica la posición del elemento medio.
d) Si el valor de la posición M es igual al valor T, indica que el valor fue encontrado y se retorna M.
e) Si el valor de la posición M es menor al valor T, $L = M + 1$ y se vuelve al paso b).
f) Si el valor de la posición M es mayor al valor T, $R = M - 1$ y se vuelve al paso b).

Un ejemplo de este tipo de algoritmo sería el que se observa en la Figura 2.5, la cual posee un arreglo ordenado con los elementos [2, 9, 11, 15, 28, 33, 40, 47, 51, 64, 76, 77, 82, 85, 94]. Se desea validar la existencia del valor 76. Como primer paso se crean las variables L, R y M, las cuales tienen el valor 0, 14 y 7, respectivamente. Se verifica el elemento que se encuentra en la posición 7, la cual es 47. Dado que es menor a 76, se modifica $L = 8 \leq R$ y se vuelve a calcular $M = 11$. Puesto que dicha posición del arreglo contiene el valor 77, se modifica $R = 10 \geq L$, generando un nuevo valor de $M = 9$, el cual contiene el 64, por lo que se vuelve a generar un cambio en $L = 10 \leq R$, lo que implica que $M = 10$, donde dicha posición contiene el elemento buscado, finalizando así la ejecución del algoritmo, sin antes retornar el valor M, dado que indica que en esa posición del arreglo, se encuentra el valor deseado.

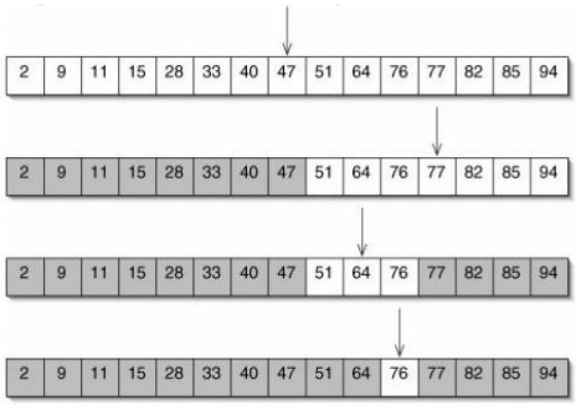


Figura 2.5. Ejemplo ilustrativo de Binary Search. [43]

Algoritmo 6: Pseudocódigo Binary Search

Input: Array, valor, l, r
Output: Indicar si el elemento fue encontrado en Array
l ← 0
r ← size - 1
Mitad ← $\frac{l+r}{2}$
if (*Array[mitad]* ! = *valor* and $l \geq r$) **then**
    Indicar que el elemento no fue encontrado
if (Array[mitad] = *valor*) **then**
    Indicar que el elemento fue encontrado
else if (*valor* > *Array[mitad]*) **then**
    Binary Search (Array, valor, mitad + 1, r)
else
    Binary Search (Array, valor, l, mitad - 1)

2.2.2 Binary Search Tree

Es un árbol binario balanceado que cumple que el subárbol izquierdo de cualquier nodo (si no se encuentra vacío) contiene valores menores que el que contiene dicho nodo, y a su vez, el subárbol derecho (si no se encuentra vacío) contiene valores mayores. Un punto para tener en cuenta es el proceso de construcción del árbol, dado que requiere que se cumpla la propiedad antes mencionada, lo que produce un trabajo adicional en comparación al anterior algoritmo de búsqueda. Los Algoritmos 7 y 8 detallan en mayor medida la función de creación del árbol (Insert) y la función de búsqueda (Search). Posee una complejidad búsqueda $O(log(n+1))$, siendo n la cantidad de elementos del árbol [17, 18, 19].

Su estructura básica se encuentra compuesta por:

a) Crear el árbol binario en base a los elementos del arreglo mediante función *Insert*.
b) La búsqueda comienza por el nodo raíz y continua mientras el nodo actual en el que se encuentre contenga un valor distinto a *null*. En el caso de no poder continuar, retornar *FALSE*.
c) Si el valor del nodo actual es igual al del elemento buscado, finaliza la ejecución y se retorna *TRUE*.
d) Si el elemento es menor que el nodo actual, continúa la búsqueda por el subárbol izquierdo y se vuelve al paso b).
e) Si el elemento es mayor que el nodo actual, continúa la búsqueda por el subárbol derecho y se vuelve al paso b).

Un ejemplo demostrativo es el que se visualiza en la Figura 2.6, la cual contiene un árbol binario que fue creado a partir de un arreglo con los elementos [6, 1, 3, 7, 4, 13, 8, 10, 14]. Se desea verificar la existencia del elemento 6, por lo que se inicia por el nodo raíz, el cual posee el valor 8. Dado que el elemento es menor que dicho nodo, se continua la búsqueda por el subárbol izquierdo, el cual a su vez posee un valor distinto a *null*. En este caso, el nodo contiene al 3, que es menor a 6, lo que implica continuar la búsqueda por el subárbol derecho. Dado que no es *null* el subárbol derecho, se procede a revisar el valor que almacena, el cual es el elemento deseado, por lo que termina la búsqueda retornando un *TRUE*.

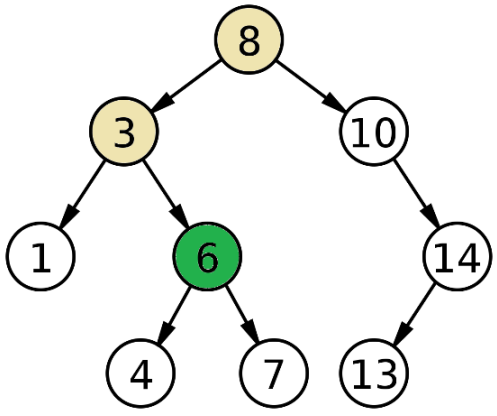


Figura 2.6. Ejemplo ilustrativo de Binary Search Tree [44]

Algoritmo 7: Pseudocódigo Insert

```
Input: valor, root
Output: Construir el árbol binario
if (root = NULL) then
        Crear un nuevo Node
        Value ← valor
        Right ← NULL
        Left ← NULL
        root  ← nuevo Node
else
        if (valor > root → Value) then
                Insert(valor, root → Right)
        else if (valor < root → Value) then
                Insert(valor, root → Left)
        else
                Valor ya se encuentra en el
                árbol
```

Algoritmo 8: Pseudocódigo Search

```
Input: elemento, root
Output: Indicar si el elemento fue
encontrado en el árbol binario
if (root ! = NULL) then
        if (root → Value = elemento) then
                Indicar que el elemento se
                encuentra en el árbol
                binario
        else if (elemento > root → Value)
        then
                Search(root → Right,
                elemento)
        else if (elemento < root → Value)
        then
                Search(root → Left,
                elemento)
        else
                Indicar que el elemento no
                se encuentra en el árbol
                binario
```

2.3 Arquitectura

Es el diseño conceptual y la estructura operacional fundamental de un computador, es decir, es un modelo y una descripción funcional de los requerimientos e implementaciones de diseño de varias partes de un computador. Los principales componentes que determinan la arquitectura que posee un computador son:

- Procesador
- Memoria primaria
- Memoria secundaria

El procesador o CPU es el encargado de interpretar las instrucciones de un programa, mediante la realización de operaciones aritméticas básicas, lógicas y externas [20]. En un equipo puede encontrarse más de un procesador con el objetivo de maximizar

el trabajo realizado. A su vez, cada procesador se compone por una memoria caché que contiene 3 niveles, cada uno con diferentes capacidades. La memoria caché es una memoria auxiliar, de gran velocidad y eficacia, en la cual se almacenan copias de los archivos y datos a los que el usuario accede con mayor frecuencia. La información almacenada puede ser de tipo inclusiva o exclusiva. En el primer caso, los datos solicitados quedan en la memoria caché de procedencia, es decir, se mantienen en dos o más niveles. Para el segundo caso, los datos solicitados se eliminan de la memoria caché de procedencia una vez transferidos al nuevo nivel. Tal como se presenta en la Figura 2.7, las memorias caché poseen una distinta distribución, presentando una relación inversamente proporcional entre capacidad y rapidez. El primer nivel, también llamado L1, está integrada al procesador del ordenador y trabaja a su misma velocidad, siendo la más rápida entre todas. Esta caché se encuentra divida en dos partes, una se encarga de almacenar las instrucciones (L1 I) y otra los datos (L1 D). El segundo nivel o L2 se encarga de almacenar datos de uso frecuente. Puede ser inclusiva y contener una copia del nivel 1, además de información extra, o exclusiva y que su contenido sea totalmente diferente de la caché L1. Su velocidad de respuesta es un poco menor con respecto al anterior nivel. Por último, se encuentra el tercer nivel o L3, la cual es compartida entre los distintos procesadores, con el objetivo de agilizar el acceso a los datos e instrucciones que no fueron localizados en la L1 y L2. Al igual que L2, puede ser inclusiva o exclusiva. Su velocidad de respuesta es menor que la de L2 [21, 22, 23]. Todo esto se encuentra contenido dentro un paquete o Package, el cual además tiene integrado otros controladores de memoria, componentes, y en ocasiones, los gráficos asociados al procesador en el caso que este disponga de aquello [33].

La memoria principal o RAM es donde se almacenan temporalmente los datos y programas que la CPU está procesando o va a procesar en un determinado momento. Esta comunicación se produce mediante el bus de datos y bus de direcciones, donde la CPU puede acceder directamente a una sección de la memoria principal sin tener que emprender un proceso de orden secuencial [24, 25].

La memoria secundaria son el conjunto de dispositivos y soportes de almacenamiento donde los datos y programas se encuentran permanente [38].

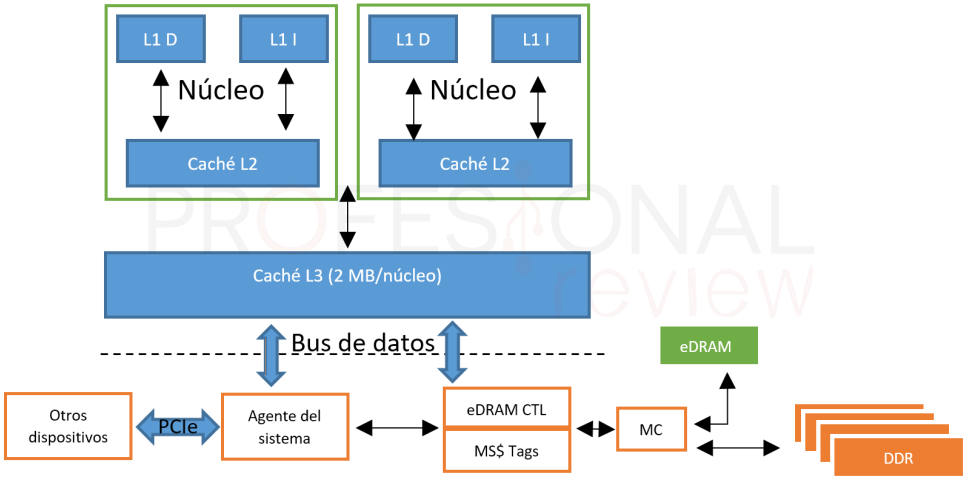


Figura 2.7. Imagen ilustrativa de la memoria caché. [45]

2.4 Perf

Es una herramienta de análisis de rendimiento en Linux, la cual permite obtener métricas acerca del funcionamiento de distintos componentes del equipo. Actualmente, es posible encontrarlo en el kernel de Linux, permitiendo medir las métricas de consumo energético de la *CPU*, del *Package* y *RAM*, la cantidad de escrituras y lecturas que ocurre tanto en la caché L1 como L3, junto a los fallos asociados a estas mismas. También otras métricas destacables son las fallas de caché, cantidad de instrucciones y ciclos de CPU, entre muchas otras.

El motivo de su elección frente a otras herramientas o dispositivos es el fácil acceso que posee, dado que únicamente es necesario contar con alguna distribución Linux que contenga un kernel[1] con versión superior o igual a la 2.6.31 [26, 27]. Además, esto implica no tener que recurrir a un costo monetario adicional, puesto que actualmente se poseen equipos que cumplen esos requerimientos. Otro factor es su alta versatilidad con respecto a las métricas que es capaz de capturar, junto a la exactitud que posee [34].

[1] Kernel: parte central de un S.O. y es el que se encarga de realizar toda la comunicación segura entre el software y hardware del ordenador.

2.5 Lenguaje de Programación

Es el medio por el cual un programador es capaz de escribir una serie de instrucciones o secuencias de órdenes en forma de algoritmos con el fin de controlar el comportamiento físico o lógico de un sistema informático. De esta manera se pueden obtener diversas clases de datos o ejecutar determinadas tareas [28]. Existen una gran cantidad de lenguajes con distintos niveles de abstracción y enfoques, entre otras características. Dado el punto anterior, la elección del lenguaje de programación utilizado se basó en un estudio [29], el cual consistió en el análisis de la energía utilizada, tiempo de ejecución y cantidad de memoria requerida para completar 10 problemas de *Computer Language Benchmarks Game*, el cual es un proyecto de software[2] que permite comparar el rendimiento de algoritmos. Los resultados indicaron que los lenguajes más eficientes con respecto a energía, tiempo y memoria requerida son C, Rust y C++, respectivamente, acotando de igual manera que no existe un lenguaje consistentemente mejor que los demás. Otro factor influyente fue la popularidad que presentan los 3 lenguajes mencionados anteriormente, en el cual Rust se encuentra por debajo de C y C++, sin siquiera ser mencionado entre los 10 más solicitados, ubicándose en la décimo sexta posición [32]. En cambio, los dos lenguajes restantes se encuentran empatados en quinto lugar, pero se optó por C++, con el propósito de crear una base comparativa que posteriormente pueda ser puesta a prueba, en especial con C.

2.6 Contenedor de software

Los contenedores consisten en agrupar y aislar entre sí aplicaciones o grupos de aplicaciones que se ejecutan sobre un mismo núcleo o kernel del sistema operativo. A su vez, son ambientes livianos que proveen a las aplicaciones con los archivos, bibliotecas y variables que necesitan para operar, reduciendo al mínimo las posibles fallas y maximizando su portabilidad [30]. Las principales diferencias que presenta con máquinas virtuales son la capacidad de ampliación y portabilidad. Dado como se observa en la Figura 2.8, un contenedor comparte el sistema operativo del equipo, en cambio una máquina virtual requiere uno adicional. Como consecuencia, un contenedor necesita una cantidad menor de recursos que la contraparte. En este caso, se utilizará Docker, el cual es un contenedor

[2] Proyecto software: proceso de gestión para la creación de un software, que encierra un conjunto de actividades.

enfocado en distribuciones Linux, pero que igualmente puede ser utilizado en otros sistemas operativos. Éste permite tener capas adicionales de abstracción y virtualización de aplicaciones, las cuales son almacenadas como imágenes [31]. El objetivo de llevar a cabo esta implementación es analizar el comportamiento energético que posee la ejecución de los algoritmos de búsqueda y ordenamiento anteriormente mencionados. A su vez, también es realizar una comparación sobre la eficiencia de energía que puede presentar este tipo de virtualización con respecto S.O nativos.

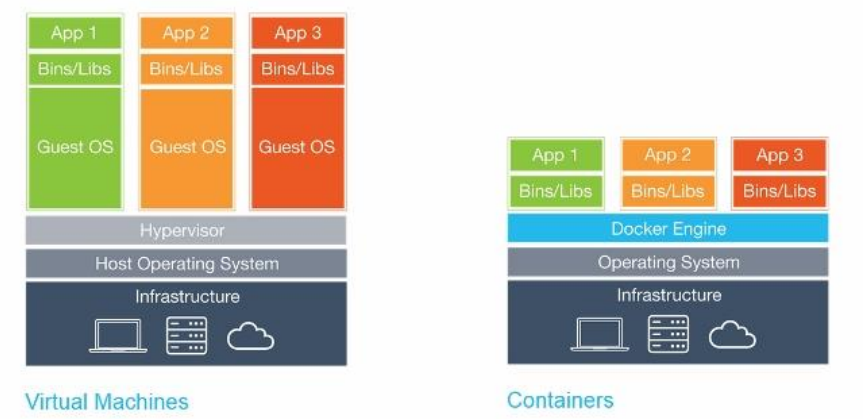


Figura 2.8. Imagen comparativa entre máquina virtual y contenedores. [46]

Capítulo 3. Metodología

En este capítulo se explican las modificaciones que fueron realizadas en los algoritmos con respecto al código base, junto a las consideraciones que se tomaron para llevar a cabo la experimentación. Además, se detalla las características de los diversos inputs utilizados y el objetivo que poseen sus variaciones. A su vez, se describe el proceso de experimentación, indicando cómo fue llevado a cabo con los distintos inputs y máquinas.

3.1 Modificaciones y consideraciones de los algoritmos

Como primer punto, se debe indicar que las versiones utilizadas de los algoritmos, y, por ende, los resultados obtenidos de los experimentos se pueden extrapolar a otras versiones de estos. Esto se debe a que solo son distintas implementaciones, pero siguen el mismo principio de algoritmo como tal. En otras palabras, los experimentos realizados pueden ser vistos como el comportamiento general o promedio de cada algoritmo mencionado anteriormente. Como segundo punto se encuentra el hecho de que para los algoritmos de Merge Sort y Binary Search se ejecutaron dos implementaciones, una recursiva y otra iterativa. Esto permite como objetivo adicional ayudar a esclarecer si el algoritmo recursivo obtiene una eficiencia energética distinta que su contraparte para esos dos algoritmos en particular. Para el caso del Merge Sort, dado que en el capítulo anterior se explicó la implementación recursiva, ahora se procederá con la iterativa, la cual se puede observar en los Algoritmos 9 y 10. Las principales diferencias se encuentran en la función "*mergesort*", la cual requiere de un método adicional que permita retornar el mínimo entre dos números, junto a la cantidad de parámetros requeridos por la función "*merge*".

Algoritmo 9: Pseudocódigo Merge Sort Iterative

```
Input: Array, Temp, low, high
Output: Array ordenado
low ← 0, high ← size - 1
for i ← 1 to high - low do
        for j ← low to high - 1 do
                from ← j
                mid ← j + i – 1
                to ← mínimo(j + 2 * i – 1,
                high)
                Merge(Array, Temp, from,
                mid, to, high)
                j ← 2 * j
        i ← 2 * i
```

Algoritmo 10: Pseudocódigo Merge Iterative

```
Input: Array, Temp, from, mid, to, high
Output: Array ordenado
k ← i ← from
j ← mid + 1
while (i ≤ mid and j ≤ to) do
        if (Array[i] < Array[j]) then
                Temp[k++] ← Array[i++]
        else
                Temp[k++] ← Array[j++]
while (i < high and i ≤ mid) do
        Temp[k++] ← Array[i]
for i ← from to to do
        Array[i] ← Temp[i]
```

Para el algoritmo Binary Search, al igual que el caso anterior, se presentó primeramente la versión recursiva, por ende, ahora corresponde la versión iterativa, la cual se encuentra presente en el Algoritmo 11. La única diferencia notable es el hecho de modificar el condicional de la función de búsqueda de un if que se encuentra en el algoritmo recursivo a un while, presente en el iterativo.

En tercer lugar, se encuentra la acción de considerar utilizar en la medida de lo posible arreglos evitando el uso de vectores, dado que los vectores al duplicar constantemente su tamaño generan una carga adicional de trabajo que se ve reflejada en tiempo de ejecución y consumo energético.

Algoritmo 11: Pseudocódigo Binary Search (parte 1)

```
Input: Array, valor, size
Output: Indicar si el elemento fue encontrado en
Array
l ← 0, r ← size – 1
while (l ≤ r) do
        mitad ← (l + r)/2
        if (Array[mitad] = valor) then
                Indicar que el elemento fue
                encontrado
```

Algoritmo 11: Pseudocódigo Binary Search (parte 2)

```
        else if (valor > Array[mitad]) then
                l ← mitad + 1
        else
                r ← mitad - 1
        Indicar que el elemento no fue
        encontrado
```

3.2 Inputs

Se utilizaron 4 categorías que presentan distintas características, con el objetivo de poder experimentar en distintos escenarios.

- Categoría 1
  - Archivos que contienen elementos con valor entre 1 y n, siendo n la cantidad de elementos.
  - El tamaño del archivo está representado por:
    - $150.000 + 100.000 * k,\ k \in \{0, 1, 2, 3, 4, 5, 6, 7\}$

- Categoría 2
  - Archivos que contienen elementos con valor entre 1 y 1024.
  - El tamaño del archivo está representado por:
    - $150.000 + 100.000 * k,\ k \in \{0, 1, 2, 3, 4, 5, 6, 7\}$

- Categoría 3
  - Archivos que contienen elementos con valor entre 1 y 65536.
  - El tamaño del archivo está representado por:
    - $150.000 + 100.000 * k, k \in \{0, 1, 2, 3, 4, 5, 6, 7\}$

- Categoría 4
  - Archivos que contienen elementos con valor entre 1 y n, siendo n la cantidad de elementos.
  - El tamaño del archivo está representado por:
    - $1.500.000 + 1.500.000 * k, k \in \{0, 1, 2, 3, 4, 5, 6\}$

La creación de cada archivo y los elementos que lo componen fue hecha de manera semi - aleatoria mediante la función rand, tal como se observa en el Algoritmo 12. Se crearon dos data set[3] para cada categoría, elaborados bajo los mismos criterios mencionados anteriormente [35]. Esto se debe a que, en principio, de esta manera debería disminuir la posibilidad de caer en casos extremos, los cuales son aquellos en el algoritmo

[3] Data set: conjunto de datos.

presenta el mejor o peor desempeño posible. En el siguiente capítulo se analizará el promedio de ambos data set que posee cada categoría, esto debido a que los resultados obtenidos fueron muy similares entre los data set ejecutados para cada categoría.

Algoritmo 12: Pseudocódigo Create File

```
Input: size, cota_superior
Output: Creación de un archivo compuesto de enteros
Crear un vector de enteros
Definir una variable de tipo archivo, definiendo su nombre de salida
Inicializar la semilla en time(NULL)
for i ← 0  to size do
        aux ← rand() % cota_superior + 1
        Añadir el elemento al vector
for i ← 0  to size do
        Agregar elemento por elemento al archivo
```

3.3 Captura de datos

Para la captura de datos, se utilizó un script en Bash. Un script es un código de programación, que contiene comandos u órdenes que se van ejecutando de manera secuencial y que se utiliza para controlar el comportamiento de un programa o para interactuar con el sistema operativo. En este caso, se utilizó una versión que ya se encontraba creada[4] para recolectar las distintas métricas capturadas por *Perf*, las cuales son almacenadas posteriormente en un archivo con formato *CSV*. A su vez, tuvo dos implementaciones distintas, uno para los algoritmos de ordenamiento y otro para los algoritmos de búsqueda, siendo la única diferencia notable el hecho de que para el segundo tipo de algoritmo, era necesario indicar la cantidad de búsquedas que se realizarían, siendo estos valores 0, 10.000, 50.000, 100.000 y 150.000. De esta manera la carga de trabajo asociada netamente por la búsqueda se encuentra en condiciones equitativas para todos los archivos, existiendo la diferencia en la cantidad de elementos en los cuales buscar. También se tuvo en consideración el número de iteraciones que tendría cada implementación, puesto que se quiere generar datos representativos del comportamiento de cada uno. Dado lo anterior, se estimó en realizar 31 iteraciones, sin antes indicar que la

[4] El script base utilizado fue desarrollado y proporcionado José Fuentes, docente del departamento ingeniería informática y ciencias de la computación de la Universidad de Concepción.

iteración 1 sería ignorada, dado que suele entregar valores alejados del promedio al ser la primera vez que se cargan datos en memoria. Por lo tanto, se considerarían las 30 iteraciones restantes para el análisis de los datos.

3.4 Procedimiento de experimentación

Los experimentos fueron llevados a cabo en dos máquinas distintas, más específicamente dos notebooks, ambas con el modo *Intel Turbo Boost* activado (el cual regula la frecuencia con la que trabaja la CPU en base a la carga de trabajo actual que esté realizando) con el objetivo de comparar el comportamiento que los algoritmos pueden tener en base a las características del equipo en el que se estén ejecutando y si los comportamientos obtenidos en tiempo de ejecución se corresponden a los de consumo energético. La primera máquina está compuesta por:

- Intel Core I5-8300h, con una velocidad base de 2.3 GHz hasta los 4.0 GHz. Consta de 4 núcleos y 8 hilos.
- 16 GB de memoria RAM DDR4 a 2400 MHz.
- 16 GB de memoria Optane.
- 1000 GB de almacenamiento interno a 7200 rpm.
- Ubuntu 20.04 LTS con versión de kernel 5.8.0-34-generic.
- Memoria caché L1 de 256 KB.
- Memoria caché L2 de 1024 KB.
- Memoria caché L3 de 8192 KB.

En cambio, la segunda máquina posee las siguientes características:

- Intel Core I7-7500U, con una velocidad base de 2.7 GHz hasta los 3.5 GHz. Consta de 2 núcleos y 4 hilos.
- 8 Gb de memoria RAM DDR4 a 2400 MHz.
- 1000 GB de almacenamiento interno a 5400 rmp.
- Linux Mint 20 con versión de kernel 5.0.4-42-generic.
- Memoria caché L1 de 128 KB.
- Memoria caché L2 de 512 KB.
- Memoria caché L3 de 4096 KB.

Para la realización de los experimentos, se contemplaron medidas que permitieran disminuir en lo posible el ruido en los resultados. Estas fueron las siguientes:

- Desconectar redes WIFI/LAN y Bluetooth del equipo.
- No ejecutar ningún programa en segundo plano.
- Ambos equipos se encuentran conectados a la corriente en todo momento, por lo que pueden ocupar su máximo potencial.

Posteriormente, cada algoritmo vio modificado en menor medida su función "*main"*, dado que era necesario incluir las métricas que son asignadas por parámetro desde el script. Para los algoritmos de ordenamiento es únicamente el tamaño del archivo, en cambio, para los algoritmos de búsqueda se agrega además la cantidad de búsquedas a realizar. Estos elementos por buscar son definidos de manera aleatoria mediante la función rand, siendo su rango de valor igual a los límites de la categoría en la cual se encuentren presentes. Además, cabe mencionar que la semilla de la función es la que posee por defecto, por ende, los elementos generados son iguales para cada máquina y Docker en cada uno de los distintos tamaños de búsqueda, según corresponda. En el caso particular de Binary Search, el cual tiene como requisito que el arreglo en el cual se busca el elemento deba estar ordenado, se utilizaron copias de los distintos archivos de entrada y a su vez, estos se ordenaron previamente. De esta manera todos los algoritmos fueron ejecutados con inputs que contuvieran los mismos elementos, pudiendo ser comparables entre si los resultados generados por cada uno.

Una vez completado lo anteriormente mencionado, se inicia la recolección de datos de cada algoritmo e implementación de manera individual, donde cada uno es compilado sin utilizar ningún comando de optimización. A continuación, es ejecutado el script, el que se encarga de recolectar los datos y entregarlos en un archivo CSV, que posteriormente es llevado a formato .xlsx o Excel. Este procedimiento se repitió para ambas máquinas utilizadas, junto a Docker, la que fue implementada únicamente en la primera máquina.

Como punto adicional, para lograr una mayor exactitud acerca del comportamiento de los algoritmos, es que se generó un nuevo proceso de recolección de datos, igual al anterior, variando únicamente en que solo se cargaron los archivos de entrada. El objetivo es poder generar valores que representen netamente a la ejecución del algoritmo como tal, sin incluir las operaciones adicionales, como lo es cargar los archivos en memoria.

Capítulo 4. Experimentación y Resultados

En este capítulo se mostrarán los resultados experimentales de los algoritmos descritos en el capítulo 2. Tendrá un enfoque centrado en el tiempo de ejecución y eficiencia energética presente en cada algoritmo. Además, se acompañará con el análisis de algunas métricas complementarias que permitan ayudar a esclarecer el comportamiento presente en las distintas categorías. Cabe recordar que en lo que respecta a energía, esta se compone por EnergyCores o energía de los núcleos, EnergyPkg o energía del paquete y EnergyRAM o energía de la RAM. Sin embargo, los resultados analizados se enfocarán en EnergyPkg, dado que su estructura se compone en mayor parte por EnergyCores, en conjunto de otros componentes, lo que permite inferir que el comportamiento de ambas será en gran medida similar. Por otro lado, dado que los distintos inputs son capaces de almacenarse en su totalidad en la RAM, su consumo no tiende a ser elevado ni indicativo de algún comportamiento en general. También es necesario indicar que las categorías 1, 2 y 3 son capaces de almacenarse individualmente dentro de la L3, dado que 850.000 elementos de tipo entero requieren 3.4 MB, inferior a los 8 y 4 MB disponibles en las máquinas 1 y 2, respectivamente. En cambio, para la categoría 4 únicamente el input de tamaño 1.500.000 es posible de almacenar en la L3 de la máquina 1 y Docker, dado que requiere 6 MB, en cambio el resto requiere mover datos entre la RAM y la L3. Otro punto, es que para discernir cuales valores son los asociados a cada máquina o contenedor, se dispuso en crear la siguiente nomenclatura: _M1 para la máquina 1, _M2 para la máquina 2 y _D para Docker.

También se abarca un análisis comparativo entre Docker y la máquina 1, con respecto al desempeño energético y tiempo de ejecución. Cabe recordar que la máquina 1 cuenta con S.O. nativo y que Docker fue instalado sobre él.

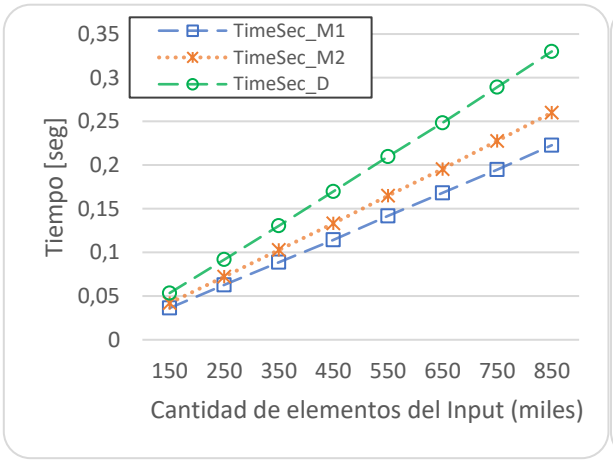
4.1 Algoritmos de ordenamiento

4.1.1 Merge Sort

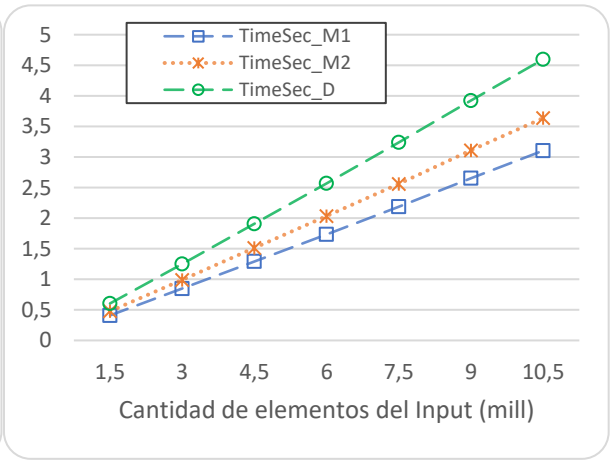
Debido a que las 4 categorías de inputs presentaron las mismas tendencias con respecto a tiempo de ejecución y eficiencia energética en las implementaciones recursivas e iterativas, se optó por analizar únicamente las categorías 1 y 4. Sin embargo, pueden existir menciones específicas a alguna de las categorías restantes.

4.1.1.1 Merge Sort Recursivo

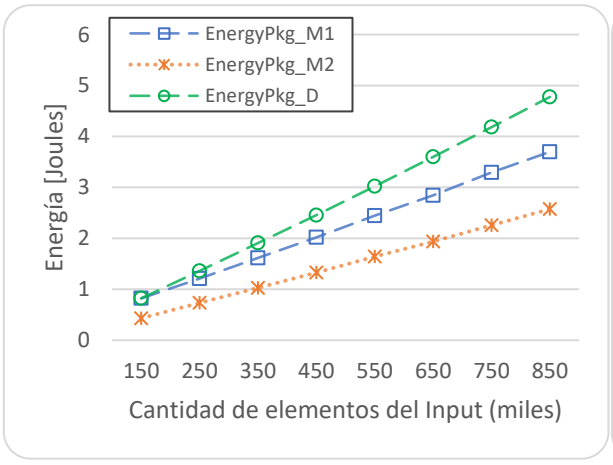
Con respecto a las métricas principales, la ejecución realizada en Docker presentó los resultados menos eficientes. En primer lugar, el tiempo promedio extra requerido fue de un 48% en comparación a la máquina 1 y de un 26.5% con respecto a la máquina 2. Para el caso de EnergyPkg, se presenta una situación similar, puesto que el consumo adicional promedio que requirió fue de un 33% en la relación a la máquina 1 y un 86% a la máquina 2. Aunque para el input más pequeño, tanto Docker como la máquina 1 presentan valores con una diferencia mínima entre ambos (*Figuras 4.1.A, 4.1.B, 4.1.C y 4.1.D*).



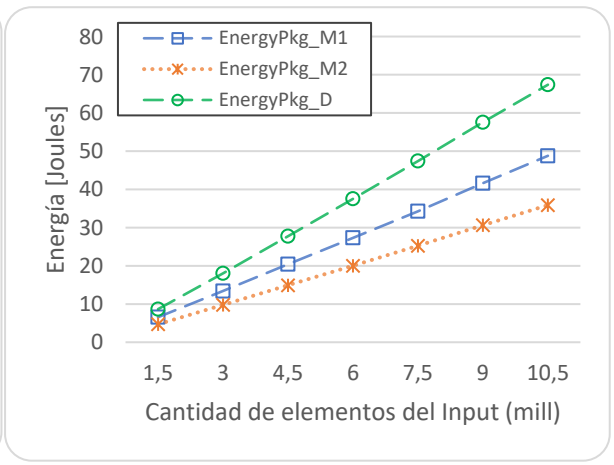
(A). Tiempo de ejecución para la categoría 1.



(B). Tiempo de ejecución para la categoría 4.



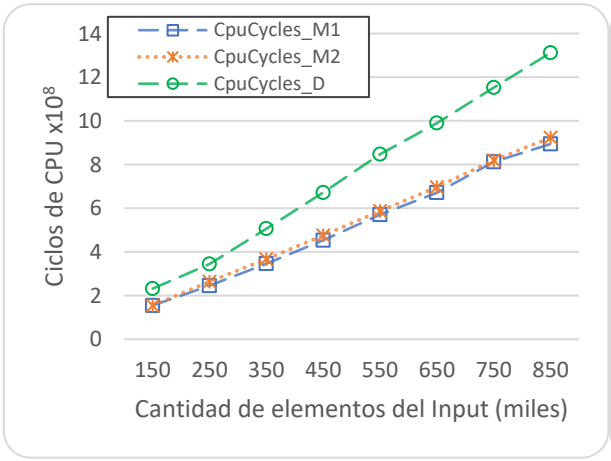
(C). Consumo energético del paquete para la categoría 1.



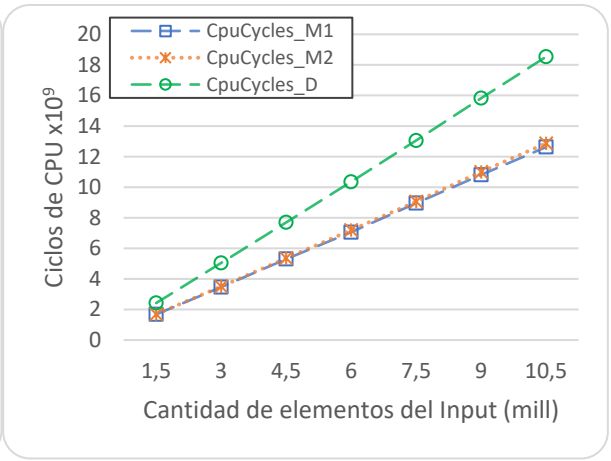
(D) Consumo energético del paquete para la categoría 4.

Figuras 4.1: resultados experimentales sobre el tiempo de ejecución y consumo energético del paquete para el algoritmo Merge Sort Recursivo en las categorías 1 y 4.

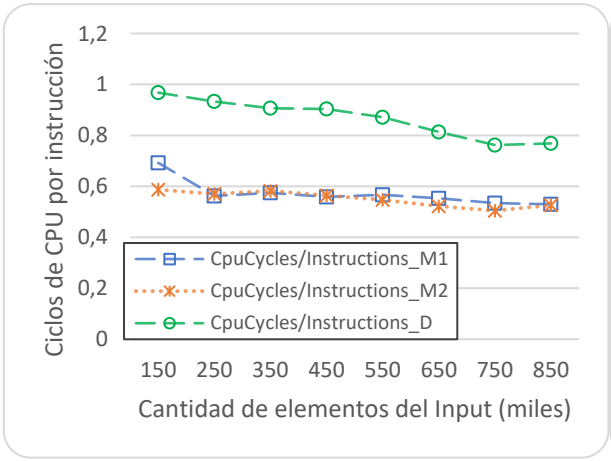
Dentro de las posibles métricas que podrían explicar el comportamiento de Docker, se encuentran los ciclos de CPU, dado su alta correlación con respecto al tiempo de ejecución y la energía requerida (*Figuras 4.5.M y 4.5.N*). Con respecto a la máquina 1 y 2, Docker se vio forzado a requerir un 46% y 43% adicional, respectivamente (*Figuras 4.2.E y 4.2.F*). Esto deriva en que la cantidad de ciclos de CPU por instrucción aumente, por lo que para ejecutar la misma cantidad de instrucciones, Docker utiliza una mayor cantidad de tiempo. Una particularidad presente en esta métrica es que para la categoría 1 se visualizan valores prácticamente decrecientes, sin embargo, a medida que crece el input, estos se estabilizan, tal como se observa para la categoría 4 (*Figuras 4.2.G y 4.2.H*). Esto permite inferir que a medida que aumenta el tamaño del archivo de entrada, los ciclos de CPU por instrucción no se incrementan en la misma medida.



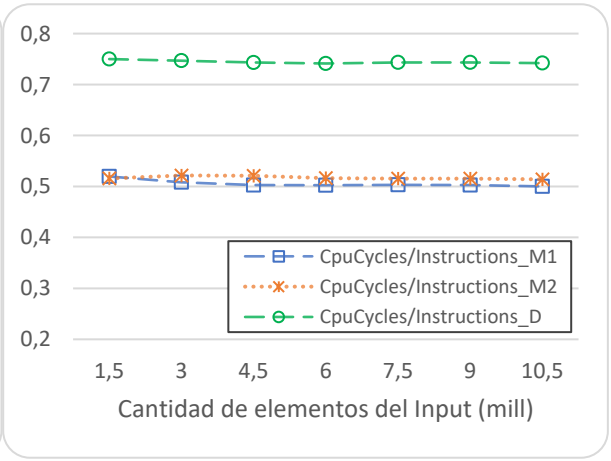
(E) Ciclos de CPU para la categoría 1.



(F) Ciclos de CPU para la categoría 4.



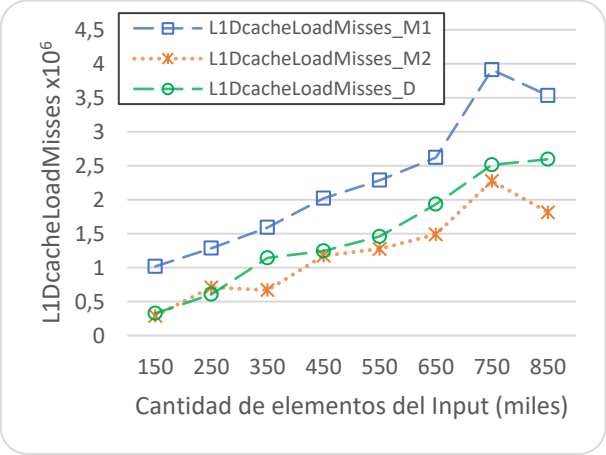
(G) Ciclos de CPU por instrucción para la categoría 1.



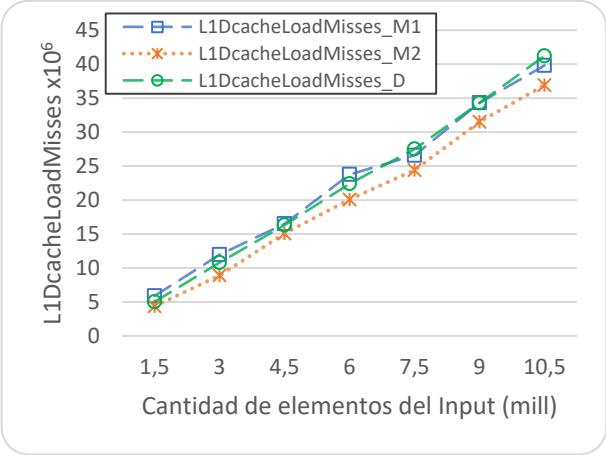
(H) Ciclos de CPU por instrucción para la categoría 4.

Figuras 4.2: resultados sobre los ciclos de CPU y ciclos de CPU por instrucción para el algoritmo Merge Sort Recursivo en las categorías 1 y 4.

Otra métrica que posee una alta correlación con la energía requerida por el algoritmo son las fallas en las cargas de datos de caché en la L1 o L1DcacheLoadMisses (*Figuras 4.3.I y 4.3.J*). Presenta una peculiaridad para las categorías 1, 2 y 3, donde la máquina 1 es la que posee más fallas, siendo 48% menos eficiente que la máquina 2 y 39% con respecto a Docker. Eso se presenta como un suceso único dentro de las métricas relacionadas a las fallas que ocurren en la L1 y L3, dado que no se repite en ninguna otra que abarque las categorías mencionadas anteriormente. Sin embargo, para la categoría 4, Docker obtiene valores muy similares con respecto a la máquina 1, dejando poco margen entre ambos. Una posible explicación podría ser el hecho de que la máquina 1 al estar ejecutándose en un S.O. nativo, la cantidad de datos que llegan a la L1D es superior a la de Docker. Esto deriva en la posibilidad de generar una mayor cantidad de fallas en la L1D, en cambio, Docker solo ejecuta lo necesario para operar, lo que en parte disminuye la cantidad de fallos.



(I) Categoría 1.

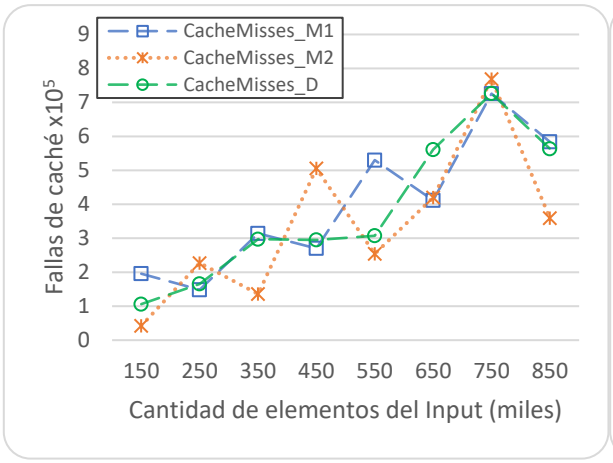


(J) Categoría 4.

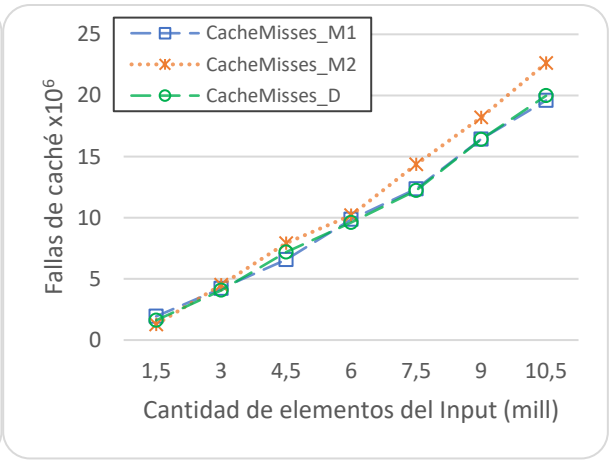
Figuras 4.3: L1DcacheLoadMisses para el algoritmo Merge Sort Recursivo en las categorías 1 y 4.

Una acotación que suele hacerse presente en las métricas relacionados a las fallas que ocurren en la L1 y L3, como podría ser CacheMisses entre otras, es que los valores obtenidos para las categorías 1, 2 y 3 suelen ser erráticos. En cambio, para la categoría 4 son directamente proporcionales entre la cantidad de elementos del input y el valor generado. Es decir, a medida que aumenta la cantidad de elementos a ordenar, las métricas

igualmente lo hacen. Un ejemplo de aquello es el que se puede observar la *Figura 4.4.K y Figura 4.4.L*.



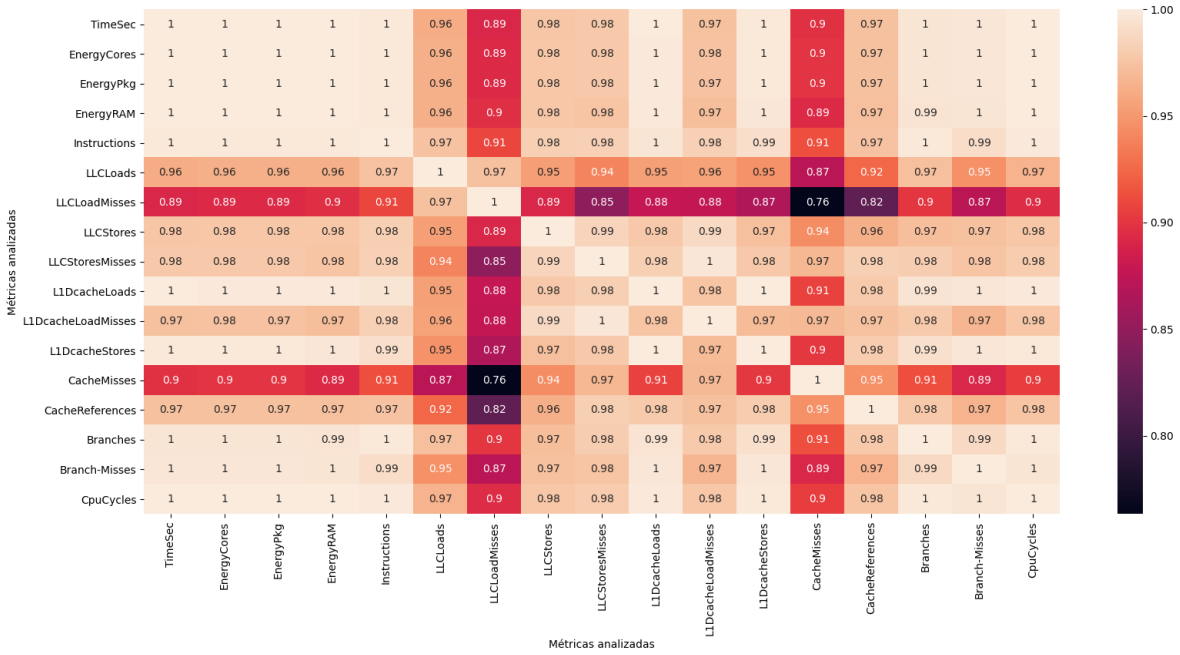
(K) Categoría 1.



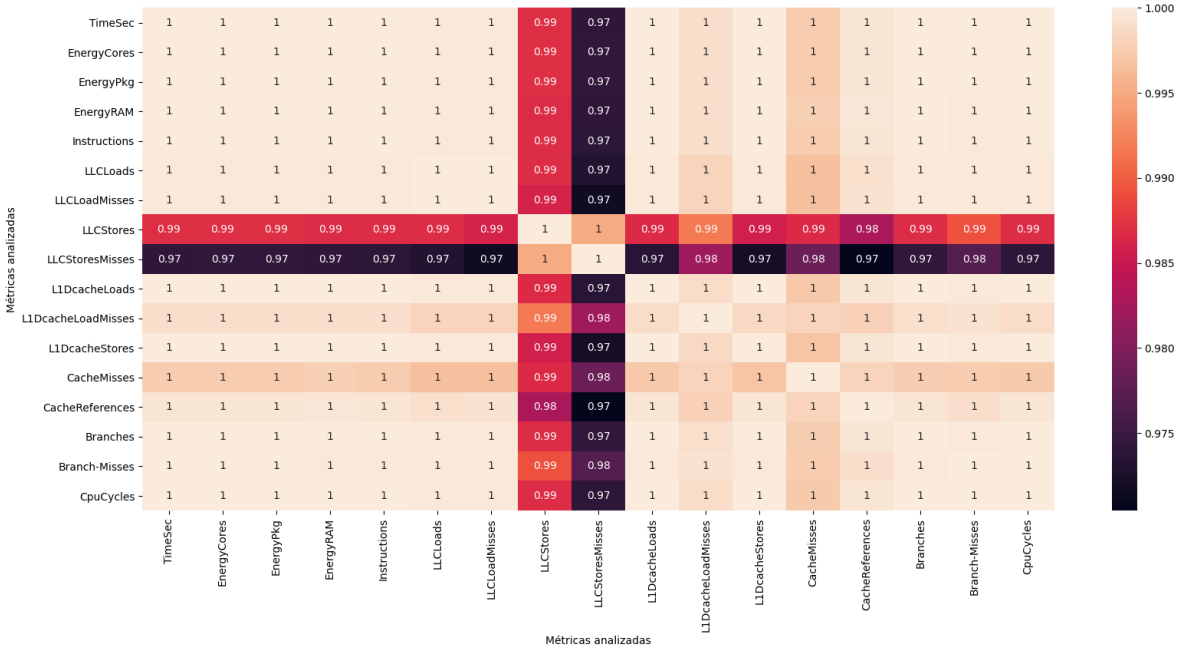
(L) Categoría 4.

Figuras 4.4: fallas de caché para el algoritmo Merge Sort Recursivo en las categorías 1 y 4.

En términos de eficiencia energética, los resultados obtenidos por la máquina 2 fueron los más eficientes, puesto que presentó en promedio un 47% menos de consumo con respecto a su par más cercano, siendo este la máquina 1. Sin embargo, en lo que concierne al tiempo de ejecución, ocurre lo contrario, dado que la máquina 1 es la que ostenta el título de más eficiente, como se visualiza en las *Figuras 4.1.A, 4.1.B, 4.1.C y 4.1.D*. Esto en principio podría deberse principalmente por las frecuencias boost y cantidad de núcleos que poseen los procesadores de ambas máquinas. Específicamente, la máquina 1 al poseer frecuencias más altas, puede disminuir el tiempo de ejecución. Caso contrario ocurre con la máquina 2, que al tener un boost reducido requiere un tiempo de ejecución superior al de la máquina 1 pero con menor consumo energético.



(M) Categoría 1.

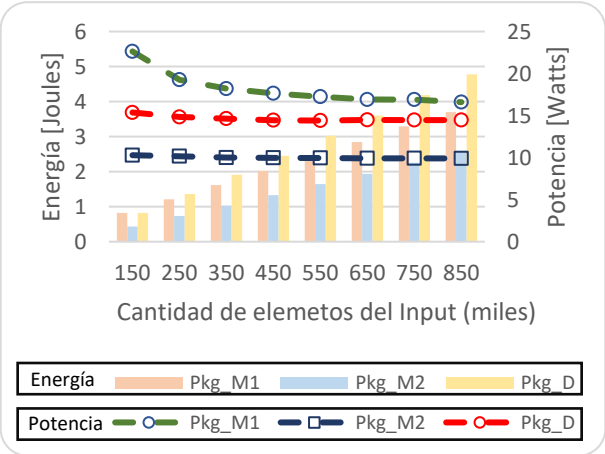


(N). Categoría 4.

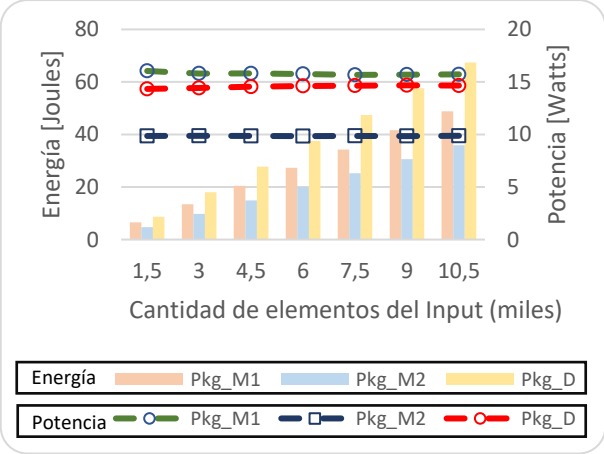
Figuras 4.5: matrices de correlación[5] para el algoritmo Merge Sort Recursivo en las categorías 1 y 4.

[5] Obs.: existen celdas con los mismos valores que presentan distintas tonalidades, debido a que hay diferencias de milésimas que por temas de espacio y legibilidad no se incluyeron.

Por último, la potencia o consumo de energía por segundo que requiere cada ejecución plantea una situación adversa entre la máquina 1 y Docker (*Figuras 4.6.O y 4.6.P*). Esto se debe a que la máquina 1 requiere un menor consumo energético, pero posee un mayor gasto por segundo, caso contrario ocurre con la ejecución en Docker. Su origen puede tener relación con el hecho de que la tasa de crecimiento del tiempo es menor que el de la energía para la máquina 1. Notar que al igual que como ocurre con los ciclos de CPU por instrucción, a medida que crece el input, la potencia presente en ambas máquinas y contenedor tienden a ser estables. En cambio, para archivos de tamaño de entrada como la categoría 1, propenden a ser decrecientes.



(O). Categoría 1.

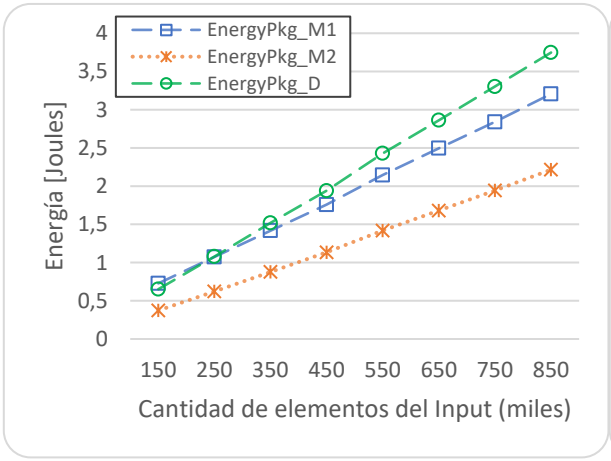


(P). Categoría 4.

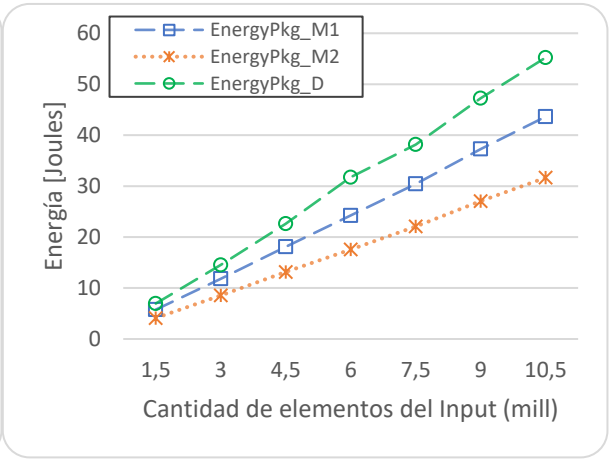
Figuras 4.6: energía y potencia del paquete para el algoritmo Merge Sort Recursivo en las categorías 1 y 4.

4.1.1.2 Merge Sort Iterativo

En términos de tiempo, no existe variación considerable con respecto a su contraparte recursiva. Sin embargo, la energía del paquete para el input más pequeño de la categoría 1 presenta una mayor eficiencia por parte de Docker con respecto a la máquina 1, siendo aproximadamente un 11%. En cambio, para el resto de los inputs de la categoría 1 y 4, se observa que Docker requiere un mayor consumo (*Figuras 4.7.A y 4.7.B*).



(A). Energía del paquete para la categoría 1.



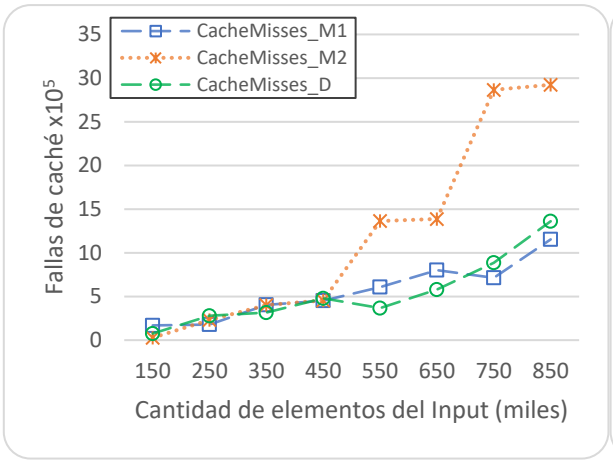
(B). Energía del paquete para la categoría 4.

Figuras 4.7: resultados sobre la energía del paquete del algoritmo Merge Sort Iterativo para las categorías 1 y 4.

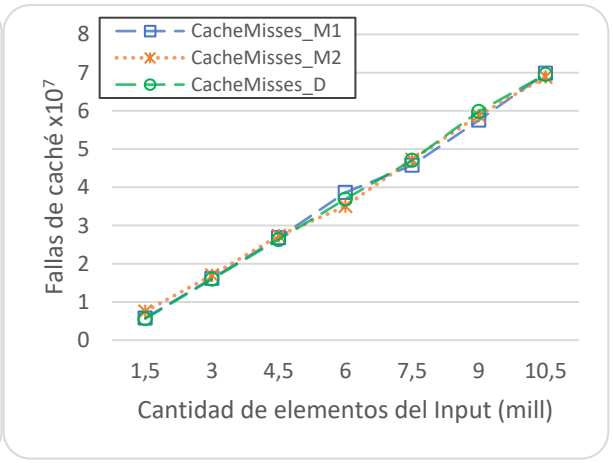
Al igual que en la implementación recursiva, las fallas de caché poseen una característica peculiar para las categorías de inputs más pequeño, es decir, desde la 1 hasta la 3. En este caso, la máquina 2 presenta valores muy altos entre 550.000 y 850.000 elementos, siendo superior en promedio un 58% en comparación a la máquina 1 y un 63% con Docker. Para la categoría 4 vuelve a ocurrir de que se observan valores estables a medida que el tamaño del input crece, por lo que la anormalidad solo ocurre en las categorías mencionadas anteriormente (*Figuras 4.8.C y 4.8.D*). En general todas las métricas poseen una alta correlación con las fallas de caché, sin embargo, hay 4 que presentaron valores levemente altos para la máquina 2. Estas son la cantidad de instrucciones, las LLCStoresMisses o fallas de carga en la L3 y la cantidad de condicionales ejecutadas por el código o branches (*Anexo 1*). Aunque los valores provistos por estas métricas no se asemejan del todo a lo ocurrido en las fallas de caché, dada su alta correlación es posible inferir que pueden ser parte de la explicación del suceso. Primeramente, la cantidad de instrucciones entre los inputs que presentan dicha condición posee entre un 6 a 8% adicional en comparación a la máquina 1 y Docker. Por otro lado, las LLCStoresMisses obtienen hasta un 36% extra de fallas en la máquina 2, situando al input de tamaño 850.000 como el más representativo de aquello. Por último, los branches generaron entre un 8 a 12% adicional, lo cual en conjunto a lo anteriormente mencionado, pueden ser uno de los principales factores de dicho comportamiento.

Con respecto a la correlación presente entre las distintas métricas, estas poseen valores inferiores a comparación de la anterior implementación para la categoría 1. Sin embargo, no son diferencias abismales, puesto que el nivel de correlación sigue siendo alto (*Figura 4.8.E*).

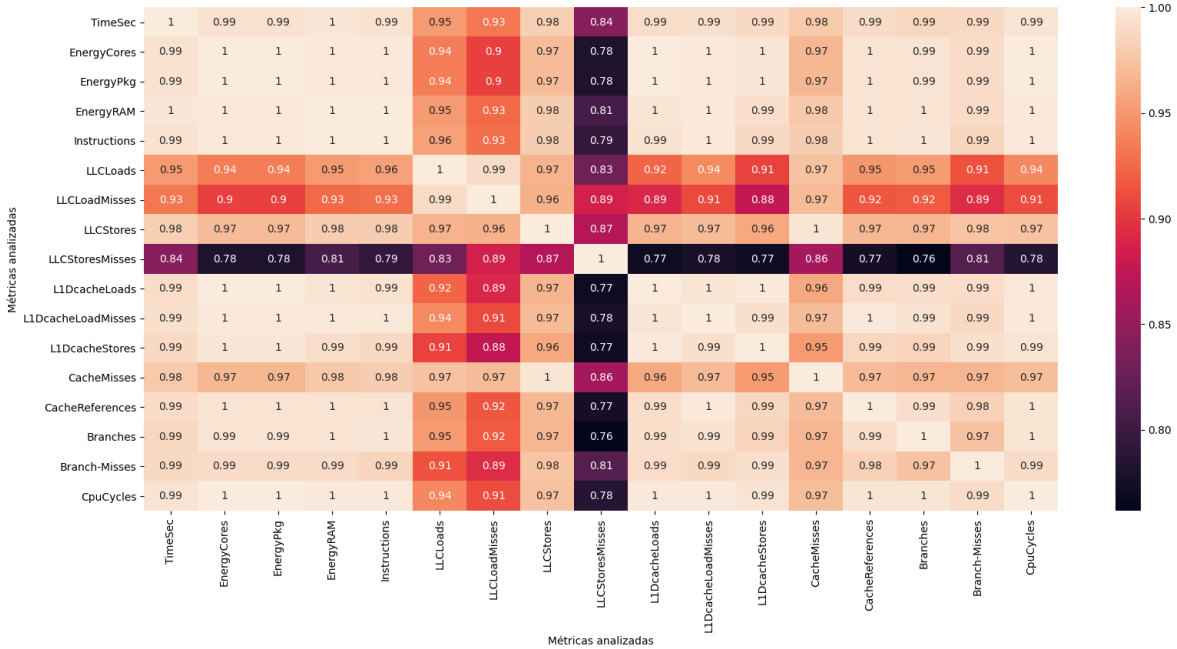
El resto de las métricas no presenta variaciones relevantes con respecto a su contra parte recursiva, por lo que no se considera destacable la mención de alguna en particular.



(C). Fallas de caché en la categoría 1.



(D). Fallas de caché en la categoría 4.



(E). Matriz de correlación del algoritmo Merge Sort Iterativo para la categoría 1.

Figuras 4.8: matriz de correlación para la categoría 1 y fallas de caché para el algoritmo Merge Sort Iterativo en las categorías 1 y 4.

4.1.2 Counting Sort

Una peculiaridad presente con *Perf*, es el hecho que para las categorías 1, 2 y 3 presentó dificultades al momento de capturar valores para ciertas métricas, como los ciclos de CPU. Debido a que no fue capaz de obtener los valores para los inputs 150.000, 250.000 y 350.000 para ambas máquinas y Docker, lo que en principio podría deberse a que los valores generados son muy pequeños para que los capture *Perf*[6]. Este mismo problema se presentó en las fallas de caché, referencias a caché, branches, entre otras, por lo que se optó por analizar únicamente aquellos tamaños de inputs que presenten valores para ambas máquinas y Docker.

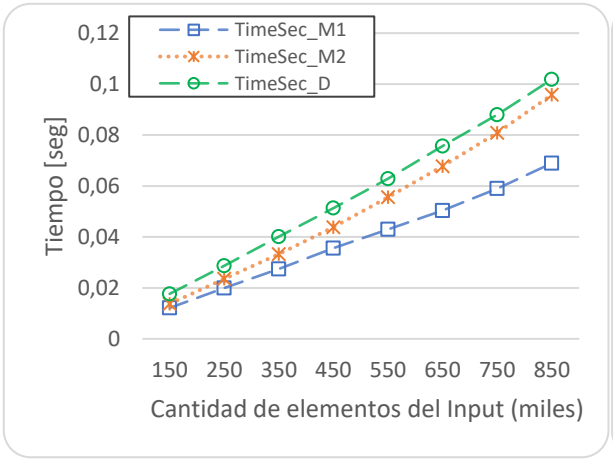
Los valores obtenidos en Counting Sort presentan dos tendencias para el apartado de tiempo y energía, donde la primera abarca las categorías 1, 2 y 3 y la segunda la categoría restante. Por ende, se decidió analizar las categorías 1 y 4 con el objetivo de indicar las diferencias presentes en las métricas mencionadas anteriormente, sin embargo, pueden existir menciones específicas a la categoría 2 y 3 de ser oportunas.

Para la primera categoría, existe una tendencia muy similar a la que fue descrita en Merge Sort Recursivo. El tiempo de ejecución no presentó variaciones, siendo Docker la implementación menos eficiente al requerir en promedio un 13% extra en comparación a la máquina 2 y un 31% con respecto a la máquina 1. Por su parte, la EnergyPkg evidenció un acontecimiento en el intervalo 150.000 - 350.000, donde la máquina 1 presentó un mayor consumo que Docker (*Figuras 4.9.A y 4.9.B*). Un problema presente, es que al existir varias métricas en las cuales no fue posible obtener valores, es complejo determinar comportamiento, debido a que ocurre justo en los tamaños de inputs que presentan dicho problema. Con respecto a esto, no es posible determinar cuál implementación es más eficiente en base al promedio, el cual es el que se encuentra representado en los gráficos. Esto se debe a que al calcular la desviación estándar en los resultados de la máquina 1 y Docker y posteriormente realizar una comparación entre ambos, se observa que existen segmentos en común. Por lo tanto, no existe una predominancia exclusiva sobre eficiencia energética de Docker sobre la máquina 1 en el intervalo 150.000 a 350.000 ni viceversa. Sin embargo, la máquina 1 domina a Docker para el resto de los tamaños de inputs.

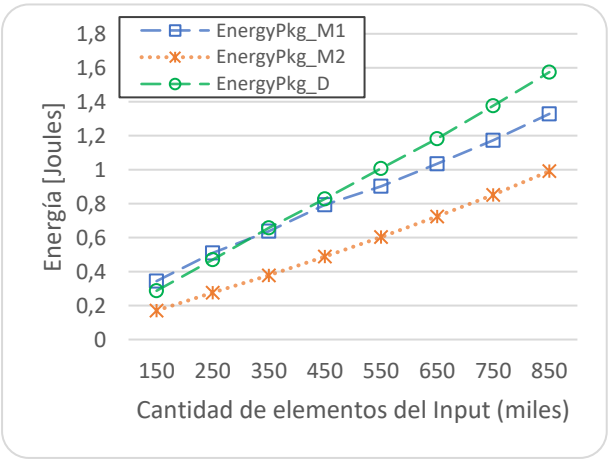
Por otro lado, la categoría 4 presenta una variación para el tiempo de ejecución, debido a que la máquina 2 es la menos eficiente, superando a Docker en comparación al resto de categorías (*Figuras 4.9.C y 4.9.D*). A priori, la única métrica que presenta valores

[6] Entregaba el mensaje <not-counted>, que indica que no fue capaz de capturar el valor.

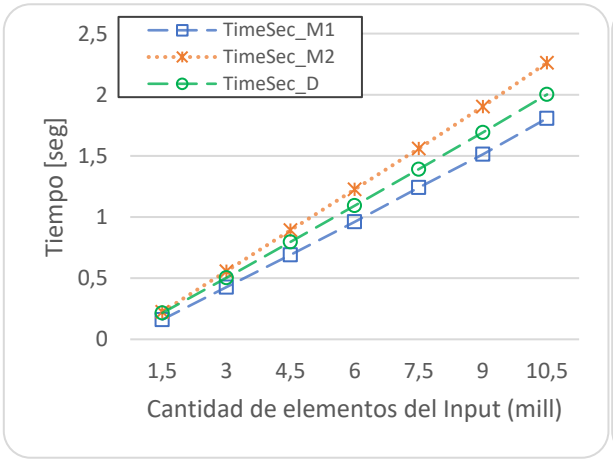
que podrían explicar dicho comportamiento son las LLCLoadMisses o fallas de carga de datos en la L3, las cuales en la máquina 2 son entre un 3% y 23% más elevadas comparadas a la máquina 1 y Docker, respectivamente. Aunque posee un nivel de correlación muy alto con respecto al tiempo de ejecución (*Anexo 2*), para el resto de las categorías ocurre un hecho similar en el cual la máquina 2 genera resultados semejantes e incluso más altos abarcando los distintos tamaños de inputs. Sin embargo, esto no se ve reflejado en su tiempo de ejecución, debido a que la máquina 2 no es menos eficiente que Docker para dichas categorías en ningún tamaño del input.



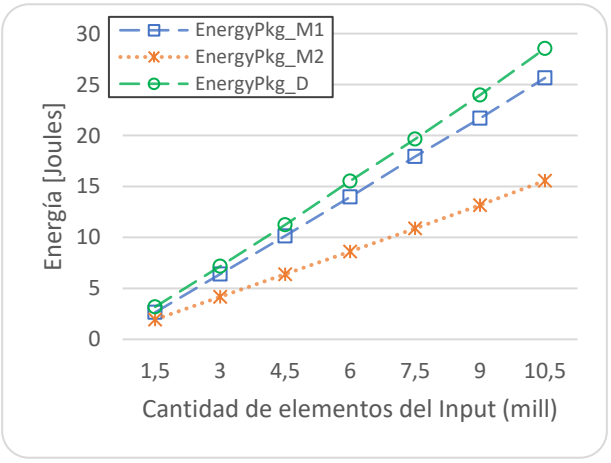
(A). Tiempo de ejecución para la categoría 1.



(B). Energía del paquete para la categoría 1.



(C). Tiempo de ejecución para la categoría 4.

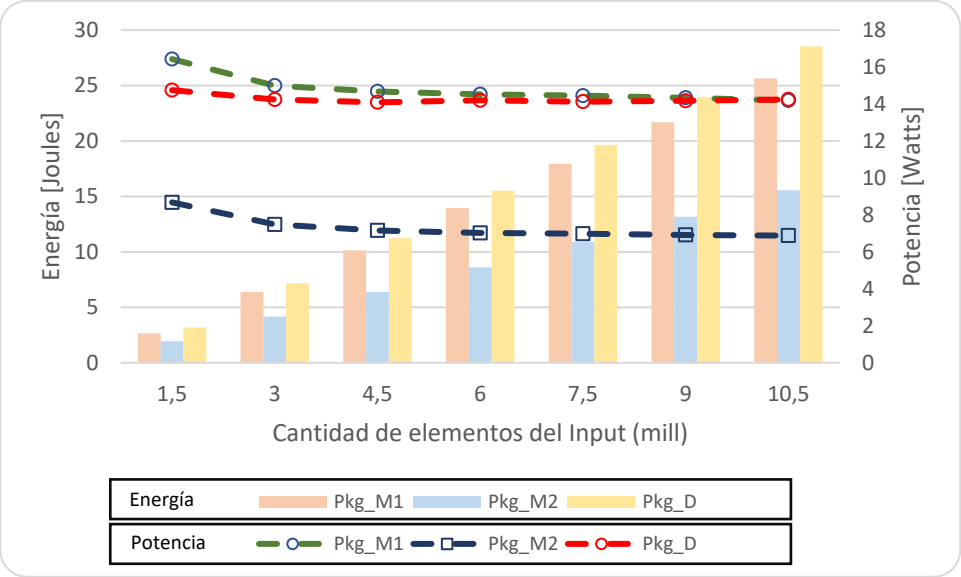


(D). Energía del paquete para la categoría 4.

Figuras 4.9: Tiempo de ejecución y energía del paquete para el algoritmo Counting Sort considerando las categorías 1 y 4.

Una peculiaridad presente únicamente en la categoría 4, es el hecho de que el consumo de energía por segundo entre la máquina 1 y Docker es muy similar. Comienza la

máquina 1 requiriendo un 12% extra hasta llegar a necesitar un 0.23% menos que Docker, dejando una diferencia casi inexistente entre ambos (*Figura 4.10.E*). Esto permitiría a Docker tener ventaja respecto a la máquina 1 en aquellos dispositivos con baterías en los cuales se realiza trabajo simultáneo o en segundo plano, debido a su menor consumo por segundo.



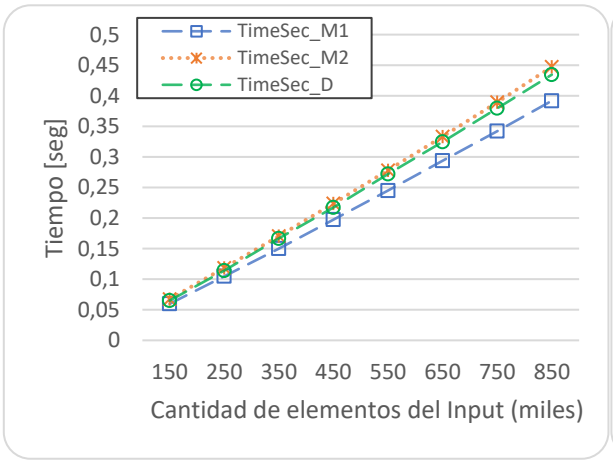
(E). Categoría 4.

Figuras 4.10: energía y potencia para la categoría 4 de Counting Sort.

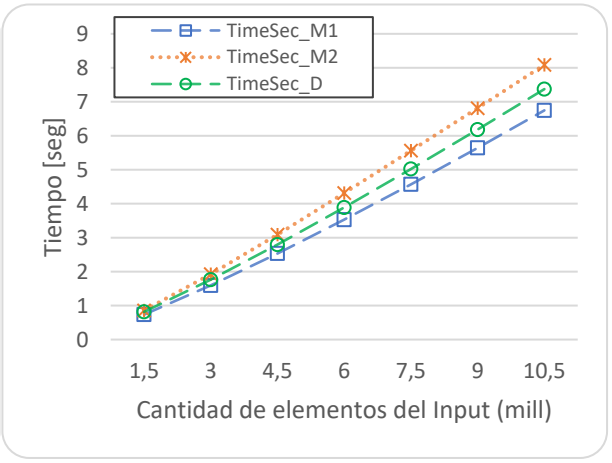
4.1.3 Heap Sort

Este algoritmo presentó dos casos distintos, el primero abarca las categorías 1, 2 y 3 y el segundo únicamente la categoría 4, por lo que se analizarán la primera y última categoría. Para el primer caso, la máquina 2 fue la menos eficiente en términos de tiempo, dado que fue en promedio un 3% superior con respecto a Docker, permitiendo que la máquina 1 sea la más eficiente (*Figuras 4.11.A y 4.11.B*). Esto se puede comprobar al revisar la desviación estándar entre ambas implementaciones, debido a que no comparten ningún intervalo en común. Por el ámbito de la energía, se presentó que entre 150.000 y 350.000 elementos, la máquina 1 obtuvo un mayor consumo, en cambio, para el resto de los tamaños de inputs fue superado por Docker. Por otro lado, la máquina 2 se mantuvo en todo momento con el menor consumo energético. Cabe señalar en lo que respecta a energía, que la diferencia en promedio entre la máquina 1 y Docker es de un 13%, 5% y 1% respectivamente para los 3 primeros tamaños de inputs. Sin embargo, únicamente para

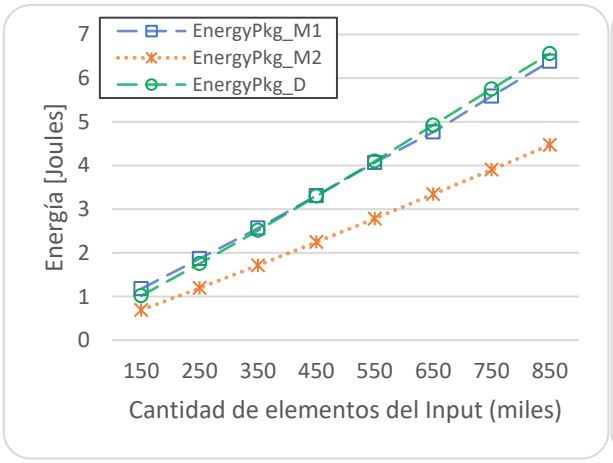
150.000 y 250.000 existe una superioridad de eficiencia energética exclusiva de Docker con respecto a la máquina 1 debido a que no existe intervalo en común en esos tamaños de inputs en base a la desviación estándar. Con respecto al segundo caso provisto por la categoría 4, este es muy similar a lo mencionado anteriormente. En relación con el tiempo de ejecución obtenido por la categoría 4, la máquina 2 aumentó aún más en promedio la diferencia con Docker, llegando a un 8%. En lo que respecta a energía, Docker es el menos eficiente necesitando en promedio un 7% más que la máquina 1. En cambio, la máquina 2 presenta los menores consumos energéticos, llegando a requerir en promedio un 25% menos que su par más cercano (*Figuras 4.11.C y 4.11.D*).



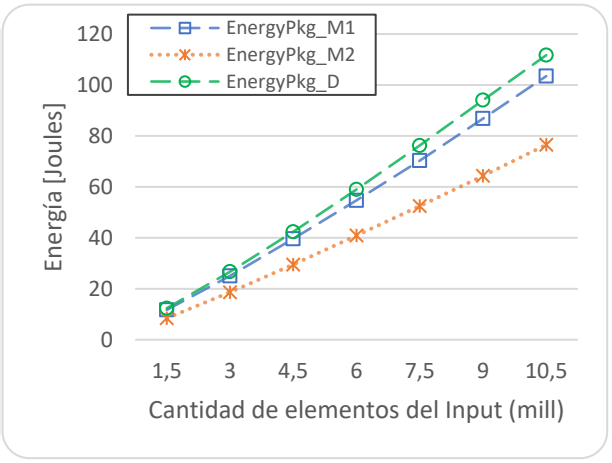
(A). Tiempo de ejecución para la categoría 1.



(B). Tiempo de ejecución para la categoría 4.



(C). Consumo energético para la categoría 1.



(D). Consumo energético para la categoría 4.

Figuras 4.11: Tiempo de ejecución y consumo energético del paquete del algoritmo Heap Sort en las categorías 1 y 4.

Para lograr entender el comportamiento descrito en los dos casos anteriores, se deben analizar por separado. En primer lugar, lo que incumbe a las categorías 1, 2 y 3 se observa que la máquina 1 obtuvo el peor rendimiento en general para todas las métricas estudiadas en los tamaños de inputs mencionados con anterioridad. Las únicas excepciones fueron los ciclos de CPU, BranchMisses, tiempo de ejecución y referencias de caché (*Anexo 3*). Cabe señalar que existe una alta correlación entre las distintas métricas con respecto a la energía (*Anexo 4*), sin embargo, los comportamientos de dichas métricas no se ven reflejadas en lo que respecta al consumo energético de la máquina 1. Esto se debe a que únicamente entre 150.000 y 250.000 existe esa variación donde la máquina 1 es la menos eficiente energéticamente. Por ende, el factor determinante puede ser la frecuencia de los núcleos del procesador o uno externo enfocado en alguna característica de la arquitectura del computador que no haya sido incluida en el presente estudio. Con respecto al tiempo de ejecución, ocurre una situación similar a lo anteriormente descrito, llegando a la misma conclusión.

Para el segundo caso que incluye únicamente a la categoría 4, al igual que el primero, presenta unas correlaciones muy altas entre las métricas de tiempo y energía con respecto al resto. En lo que involucra al tiempo de ejecución, no existen métricas que posean tendencias semejantes, únicamente similares con respecto a la eficiencia de cada máquina y Docker. Estas son las fallas de caché y LLCLoadMisses, las cuales permiten observar que la máquina 2 es la menos eficiente y la máquina 1 con Docker se encuentran prácticamente a la par. Por otro lado, solo los BranchMisses presentan tendencias similares a lo provisto por el EnergyPkg, dado que las otras métricas en las cuales Docker requiere mayor energía no se produce que la máquina 2 sea la más eficiente (*Anexo 5*).

En lo que respecta a potencia o consumo de energía por segundo, se observa la misma tendencia que en el algoritmo Counting Sort para la categoría 4, por lo que se llega a la misma conclusión mencionada.

4.1.4 Análisis comparativo de los algoritmos de ordenamiento

Para las categorías 1 y 4, se realizó un análisis comparativo entre los distintos algoritmos con el fin de determinar la eficiencia energética y la relación que poseen con el tiempo de ejecución. En primer lugar, para todos los algoritmos mencionados anteriormente la máquina 1 es la más eficiente en tiempo de ejecución y la máquina 2 en consumo energético, exceptuando casos puntuales que fueron mencionados. Debido a esto, se

realizaron gráficos comparativos para el tiempo de ejecución de la máquina 1 y el consumo energético de la máquina 2 para ambas categorías (*Anexo 6*). Los resultados indicaron que tanto para tiempo como consumo energético Heap Sort es el menos eficiente, seguido por Merge Sort Recursivo y Merge Sort Iterativo, siendo poca la diferencia entre ambos. En primer lugar se encuentra Counting Sort, el cual es más rápido que el resto y requiere un menor consumo de energía, lo cual puede verse influenciado por las características del input utilizado, es decir, fijar los límites que podrían alcanzar los elementos en términos de valor. Por otro lado, existen ejemplos en los cuales el tiempo de ejecución de un algoritmo en una de las máquinas y/o Docker no tiene directa relación con su eficiencia energética. Esto se debe a que se presentaron situaciones en las cuales alguna máquina obtenía la mayor eficiencia en tiempo de ejecución pero esto no se reflejaba en su consumo de energía. Un ejemplo concreto es el observado en el algoritmo Heap Sort para la categoría 1 y 4.

4.2 Algoritmos de búsqueda

Para el algoritmo Binary Search Tree (BST) se decidió tomar una consideración que permita realizar un análisis comparativo más equitativo con respecto al Binary Search. Esto se debe ya que para Binary Search se utilizan los mismos inputs pero ordenados, y el costo de realizar esta operación no fue incluido. Cabe señalar que el ordenamiento de los inputs fue realizado mediando la función *sort* disponible en la biblioteca algorithm.h de C++. Una consideración inicial tuvo relación con el costo asociado a la creación del árbol, el cual es incluido dentro de los valores obtenidos por las métricas, por lo cual se optó por descontar dichos valores del total. En principio, esto se realizó mediante la resta de $A - B$, donde A son los valores obtenidos de la creación del árbol junto a las búsquedas realizadas y B únicamente la creación de árbol. Sin embargo, los resultados obtenidos de esa operación mostraron un comportamiento anómalo, especialmente para la máquina 1 y Docker, debido a que en repetidas ocasiones y en todas las categorías existían valores negativos en energía y/o tiempo. Esto incluía a los distintos tamaños de elementos por buscar, desde 0 hasta 150.000 elementos. En un comienzo, una de las posibles causas es la impredecible activación del *Intel Turbo Boost*, el cual decide automáticamente subir las frecuencias de los núcleos, lo que genera en principio un menor tiempo de ejecución a cambio de un mayor consumo energético. Para poder obtener los valores que mejor representen al algoritmo, se decidió repetir la experimentación del algoritmo BST desactivando el modo turbo. Se

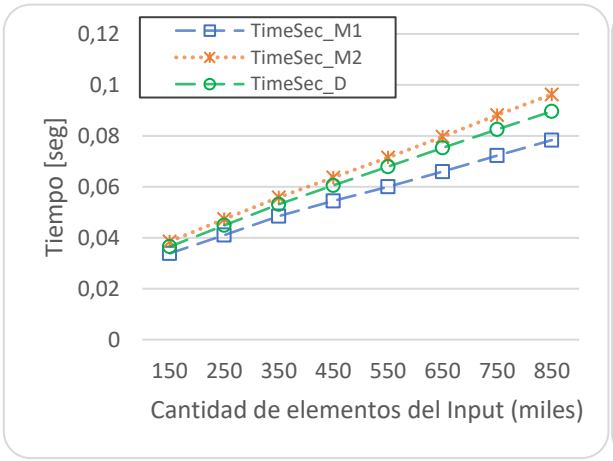
comenzó con la máquina 1 y Docker mediante la desactivación por comando: "*echo 1* | *sudo tee /sys/devices/system/cpu/intel_pstate/no_turbo*", en cambio la máquina 2 fue mediante configuración de BIOS. Esto fue un hecho excepcional, debido a que el resto de los algoritmos de búsqueda y ordenamiento tuvieron activo el *Intel Turbo Boost* al momento de ser ejecutados, sin embargo, no presentaron las tendencias anómalas descritas anteriormente. Además, un factor que limita la posibilidad de repetir todos los experimentos y análisis respectivos es el tiempo permitido para la elaboración del presente documento. Retomando a la nueva experimentación realizada, esta produjo resultados similares a los provisto por la experimentación inicial para la máquina 1 y Docker, es decir, existen tendencias anómalas. Una posible teoría es el hecho de que el *Intel Turbo Boost* no logra ser apagado del todo al no ser mediante la desactivación por BIOS. Debido a esto, se optó por repetir nuevamente la experimentación de BST pero agregando una métrica que indique el tiempo de creación del árbol, en conjunto con la inicial que captura el tiempo de toda la ejecución, incluyendo la de las operaciones. De esta manera, es posible estimar el costo de búsqueda de los elementos al determinar la proporción que ocupa la creación del árbol del total del tiempo utilizado. Esta estimación igualmente fue extrapolada al ámbito energético del algoritmo, permitiendo obtener valores que solo indiquen el costo de realizar las búsquedas y cargar los datos en memoria, al igual como ocurre con Binary Search.

4.2.1 Binary Search

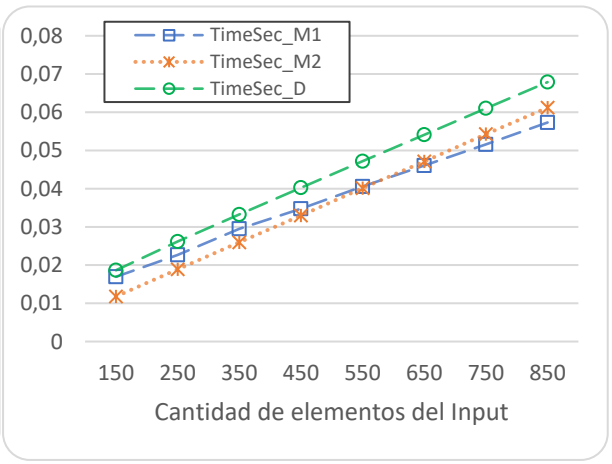
Se observó que la implementación iterativa posee dos tendencias en lo que respecta a los valores obtenidos en tiempo de ejecución y consumo energético, las cuales abarcan desde la primera hasta la tercera categoría y la restante únicamente la cuarta. En cambio, la implementación recursiva posee tres tendencias, donde la primera incluye a la categoría 1, la segunda a la categoría 4 y por último la tercera tendencia a las categorías restantes. Estas tendencias a su vez se replican para las distintas cantidades de búsquedas, permitiendo utilizar las 150.000 búsquedas como representantes del resto, sin embargo, pueden existir menciones específicas a otras cantidades de ser necesario. Para la primera y segunda tendencia se analizarán las categorías 1 y 4, respectivamente, en cambio para la tercera tendencia de la implementación recursiva se considerará a la categoría 2.

Primeramente, se evaluó las tendencias de la implementación iterativa, comenzando con la que tiene relación a la categoría 1, en la cual además de las 150.000 búsquedas se mencionará las 50.000. En términos de tiempo, la máquina 1 es la más eficiente dado que requiere entre un 10.5% a 15.4% menos que Docker y la máquina 2,

respectivamente (*Figura 4.12.A*). Sin embargo, al realizar 50.000 búsquedas ocurre que el tiempo de ejecución de la máquina 2 disminuye considerablemente, llegando a ser inferior que el de la máquina 1 para los 4 primeros tamaños de archivos de entrada. En cambio, para los 4 posteriores tamaños de archivos de entrada, la máquina 2 requiere una mayor cantidad de tiempo que la máquina 1, pero no llega a ser equiparable a lo requerido por Docker, que es superior a todos (*Figura 4.12.B*). Este fenómeno es llamativo, dado que solo ocurre con esta cantidad exacta de búsquedas. Además, se debe señalar que existe una pequeña variación entre las 3 primeras categorías acerca de la cantidad de elementos de entrada en los cuales la máquina 2 es más eficiente que la máquina 1. Estos pueden variar entre los inputs 150.000 a 350.000 o 150.000 a 450.000.



(A). 150.000 búsquedas.



(B). 50.000 búsquedas.

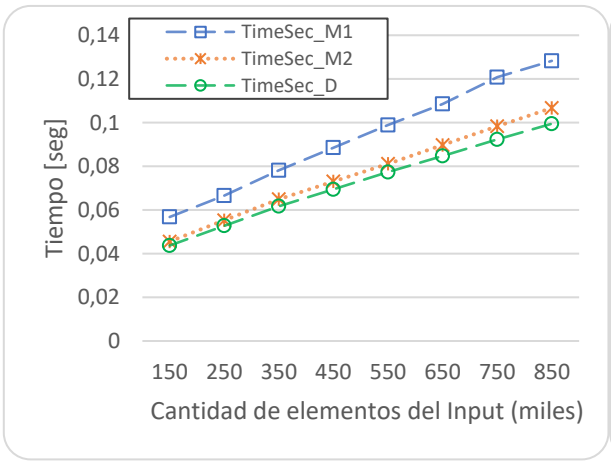
Figuras 4.12: Tiempo de ejecución del algoritmo Binary Search Iterativo para la categoría 1 con 50.000 y 150.000 búsquedas.

En cambio, lo relacionado a energía presenta un comportamiento único para el cual no existen excepciones como lo ocurrido anteriormente. Este consta que la máquina 1 es la que requiere mayor energía para su ejecución, generando que la máquina 1 sea la más rápida pero a la vez la de mayor consumo (*Anexo 18*). Por otro lado, la máquina 2 es la que posee mayor eficiencia energética, siendo superior al resto. Las diferencias porcentuales entre las distintas máquinas y Docker varía levemente entre las distintas categorías y cantidades de elementos a buscar. Sin embargo, no existen casos en los cuales el orden descrito previamente se vea alterado. Dentro de las métricas estudiadas, no existe alguna que presente valores que hagan alusión al comportamiento energético y de tiempo visto con anterioridad. Por lo tanto, es posible que factores como la frecuencia del procesador

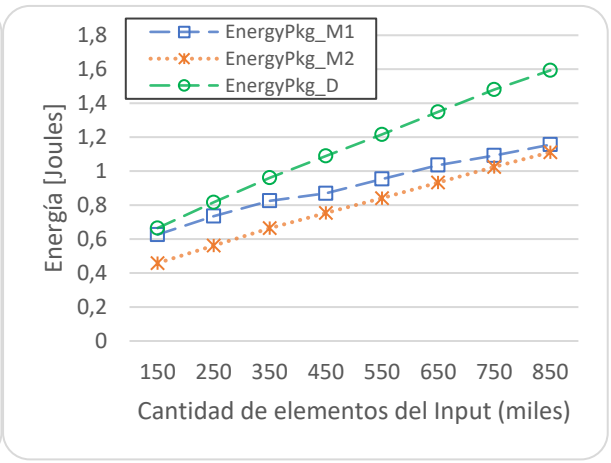
influyan en el comportamiento, e incluso, características de la arquitectura del computador que no están presentes en el estudio.

La segunda tendencia permite visualizar el comportamiento más común que se ha obtenido entre los algoritmos, tanto en tiempo de ejecución como en consumo energético. En primer lugar, el tiempo de ejecución está regido por el siguiente orden que inicia con él requiere mayor tiempo, comenzando con la máquina 2, posteriormente se encuentra Docker y luego la máquina 1, siendo esta la de mayor eficiencia. Por el lado energético, se observa que la máquina 2 obtiene el mejor desempeño, y Docker el peor, dejando en un punto intermedio a la máquina 1. Dentro de las posibles métricas que explican dicho comportamiento para el tiempo de ejecución, se encuentran las LLCLoadMisses de la máquina 2, las cuales son en promedio entre un 44 a 45% superior a la máquina 1 y Docker, respectivamente. Otra métrica que presenta comportamiento indicativo de lo mencionado con anterioridad, más específicamente en el apartado de energía son las Branch-Misses. Esta métrica reportó valores entre un 64 a 70% extra para Docker, lo que puede ser un factor relevante al momento de determinar las causas de dicho comportamiento. Esto se debe al nivel de correlación existente entre el tiempo de ejecución y las métricas mencionadas anteriormente (*Anexo 7*).

Por otro lado, ocurre una situación similar para la implementación recursiva en lo que respecta a la primera tendencia representada por la categoría 1, dado que igual ocurre un hecho particular para las 50.000 búsquedas. Como primer punto, se puede mencionar que el tiempo de ejecución varía entre un 18 a 22% de la máquina 1 con respecto a la máquina 2 y Docker, sin embargo, en esta ocasión la máquina 1 es la menos eficiente (*Figura 4.13.A*). En lo que respecta a consumo energético, se visualiza un comportamiento distinto al descrito en el anterior algoritmo. Esto se debe a que la máquina 1 presenta mayor afinidad para los primeros y últimos tamaños de inputs comparados a Docker y la máquina 2, respectivamente, sin embargo, el menos eficiente es Docker (*Figura 4.13.B*). Nuevamente, las LLCLoadMisses y Branch-Misses son las únicas métricas que permiten observar un comportamiento similar al del tiempo de ejecución y consumo energético. Otra característica en común es el comportamiento anómalo que presentan las 50.000 búsquedas debido que Docker requiere una mayor cantidad de tiempo, siendo que para el resto Docker y la máquina 2 se encuentran prácticamente a la par.



(A). Tiempo de ejecución.



(B). Consumo energético del paquete.

Figuras 4.13: Tiempo de ejecución y consumo energético del paquete del algoritmo Binary Search Recursivo para la categoría 1 con 150.000 búsquedas.

En lo que respecta a la segunda tendencia de la categoría 4, la única variación se encuentra presente en el apartado energético, el cual posee a la máquina 1 como la más eficiente, seguido por la máquina 2. Una métrica que podría explicar el comportamiento atípico por parte de dicha máquina en lo que respecta a energía, son los ciclos de CPU debido a que es la única métrica que reporta el menor valor para la máquina 1. Por último, la tercera tendencia generó un comportamiento distinto relacionado al tiempo de ejecución, debido a que la máquina 2 fue la más eficiente para los distintos tamaños de búsquedas. Esta superioridad es absoluta debido a que no existen segmentos compartidos con su par más cercano Docker. Lo anterior tiene énfasis por el hecho de que la diferencia porcentual es muy baja, siendo de apenas un 4.5% aproximadamente.

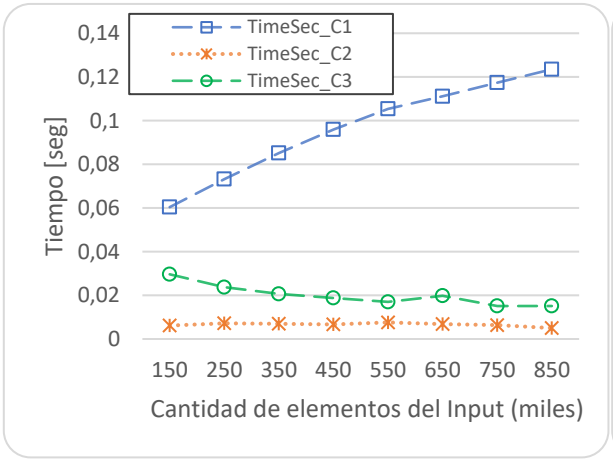
Por otro lado, la potencia requerida por el algoritmo y sus tendencias dependerá de la implementación que se lleve a cabo, debido que existen ciertas diferencias entre el algoritmo iterativo y recursivo. Una de ellas se encuentra en la categoría 1, debido a que el consumo por segundo de la máquina 1 es mayor en la implementación iterativa, en cambio en la recursiva es superado por Docker (*Anexo 11*). Por otra parte, la categoría 4 difiere en cual máquina requiere un menor consumo por segundo, dado que el más eficiente es la máquina 2 en la implementación iterativa y en la recursiva es la máquina 1 (*Anexo 12*). Esto es un tema relevante con respecto a los dispositivos que utilizan baterías, debido a que si un algoritmo requiere una menor cantidad de energía por segundo, permite una mayor cantidad de programas ejecutándose en simultáneo.

4.2.2 Binary Search Tree

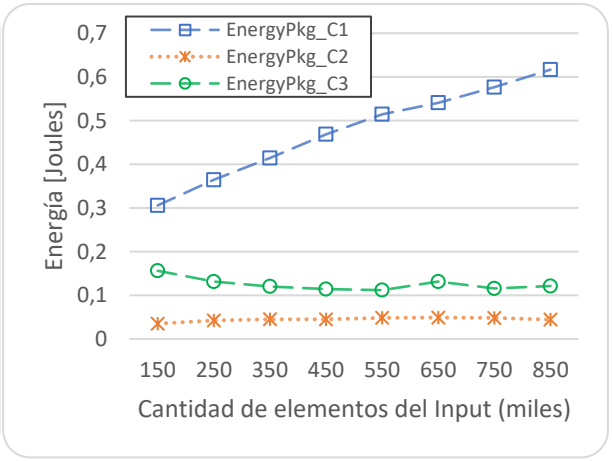
Primeramente, cabe señalar que el porcentaje de tiempo que se requiere para realizar únicamente las búsquedas oscila entre un 3 a 6% del total de tiempo de ejecución que posee el algoritmo Binary Search Tree. Por ende, los valores asociados a las métricas de energía serían una representación equivalente a los porcentajes mencionados con anterioridad. Sin embargo, existe la posibilidad que dicha representación carezca de exactitud, debido a que el consumo energético puede presentar momentos en los cuales se requiera más o menos energía que el promedio. En otras palabras, puede ocurrir que el realizar las búsquedas tenga un costo energético mayor o menor que crear el árbol, pero esto es algo que no es posible determinar con el presente estudio, así que se asumirá que el consumo es lineal durante toda su ejecución. Por otro lado, es necesario validar que el comportamiento de las métricas después de aplicar el porcentaje sea similar a lo que se obtiene al restar la ejecución total del algoritmo menos la construcción del árbol. Para aquello, se utilizará la máquina 2 debido a que fue la única que no presentó valores negativos en la resta al desactivar el *Intel Turbo Boost*. Al analizar la proporción con respecto a la resta para cada categoría y desde las 10.000 búsquedas en adelante, se obtuvo como resultado que la proporción no es representativa del algoritmo. Esto se debe a que para la mayoría de los tiempos de ejecución y consumo energético de los núcleos, paquete y RAM presentaron distintas tendencias entre la proporción y la resta. Por ende, finalmente se analizará únicamente a la máquina 2 sometida a la resta de la ejecución total menos la construcción del árbol.

Debido a que este es el único algoritmo en el cual no se podrá estudiar el comportamiento en ambas máquinas y Docker, se realizará una comparativa entre las primeras 3 categorías. En cambio, para la última categoría se normalizarán los valores de tiempo de ejecución y energía del paquete para ver sus tendencias con respecto a las cantidades de elementos a buscar comenzando desde 10.000. Se utilizará la nomenclatura _CX, donde X $\in \{1,2,3,4\}$ siendo estos los valores de la categoría que representa.

En lo que respecta al primer análisis, tanto tiempo de ejecución como consumo energético presentan las mismas tendencias para todas las cantidades de búsquedas, las que poseen relación directa con el valor máximo permitido dentro de la categoría. Esto quiere decir que la categoría 1 es la que consume una mayor cantidad de tiempo y energía, seguido por la categoría 3 y por último la categoría 2 (*Figuras 4.14.A y 4.14.B*)



(A) Tiempo de ejecución del algoritmo BST.



(B) Consumo energético del paquete del algoritmo BST.

Figuras 4.14: Tiempo de ejecución y consumo energético del paquete del algoritmo Binary Search Tree con 150.000 búsquedas.

Con relación a la categoría 4, lo que se hace presente para cada una de las métricas de tiempo y consumo energético es que a medida que aumenta el tamaño del input, tiende a bajar el tiempo o energía que se necesita por elemento buscado. Esto se debe a que en el caso del tiempo, para los 10.500.000 de elementos, se requiere en promedio un 64% menos que para el 1.500.000 de elementos. Por un lado similar y en la misma situación, la energía del paquete en promedio es un 27% menos para el input más grande con respecto al más pequeño.

4.2.3 Análisis comparativo de los algoritmos de búsqueda

Para realizar una correcta comparación, se consideran por una parte ambas implementaciones del algoritmo Binary Search para las categorías 1 y 4, dado que las tres primeras categorías presentan la misma tendencia. Por otro lado, y en deseo de incluir a Binary Search Tree se analizará en conjunto con Binary Search las mismas categorías pero solamente la máquina 2. Además, se considerará las 150.000 búsquedas de elementos dado que es donde se realiza la mayor carga de trabajo registrado en este estudio, pero pueden existir menciones a otras cantidades de ser oportuno. Cabe añadir, que para determinar los valores asociados a cada algoritmo, se dispondrá de la nomenclatura BSI

para Binary Search Iterativo, BSR para Binary Search Recursivo y BST para Binary Search Tree.

Para la categoría 1 de Binary Search, se observa que la implementación iterativa es más eficiente que la recursiva tanto en tiempo de ejecución como consumo energético. En términos de porcentaje, BSR requiere en promedio un 23% más con respecto al tiempo y un 12% en relación con el consumo de energía del paquete (*Anexo 14*). Por otro lado, la categoría 4 presenta una tendencia distinta en el apartado energético, dado que la implementación recursiva desde los 3.000.000 de elementos requiere un menor consumo que su contraparte. Para el primer tamaño de input no hay exclusividad por parte del algoritmo BSR debido a que existen intervalos comunes con BSI al determinar la desviación estándar. Cabe añadir que el tiempo de ejecución presenta la misma tendencia descrita para la categoría variando los porcentajes, dado que en promedio el BSR requiere un 19% más en la categoría 4 (*Anexo 15*).

Para la categoría 1 que incluye a BST, se observa el caso de que la máquina 2 de este algoritmo requiere un consumo inferior que ambas implementaciones de BS. Sin embargo a medida que aumenta la cantidad de búsquedas, su tiempo de ejecución llega a superar al resto (*Anexo 16*). Cabe recordar que Binary Search no cuenta con la desactivación del *Intel Turbo Boost*, por ende, no es posible afirmar que BS sea más rápido en el caso de las 150.000 búsquedas para la categoría 1. Esto se debe a que al desactivar el *Intel Turbo Boost*, se espera que el tiempo de ejecución aumente en cierta medida, lo que hipotéticamente podría equiparar los tiempos de ejecución entre ambos algoritmos. En términos de consumo energético del paquete, BST con la máquina 2 es la más eficiente para todas las cantidades de búsquedas, mientras que ambas implementaciones de BS posee prácticamente los mismos valores. La única diferencia presente es que a medida que crece la cantidad de búsquedas, la implementación recursiva tiende a requerir un 11% más en promedio. Por otra parte, la categoría 4 presenta un único comportamiento tanto para tiempo como consumo energético, el cual se compone de que BST es el más eficiente. En cambio, ambas implementaciones de BS llegan a demorar o consumir según corresponda, valores muy cercanos entre sí, lo que abarca todas las cantidades de búsquedas (*Anexo 17*). Debido a lo anteriormente mencionado, donde BS no posee *Intel Turbo Boost* desactivado, solo se puede afirmar que el tiempo de ejecución de BST para la categoría 4 debería ser inferior o más eficiente que BS. Esto dado que si se desactivara el *Intel Turbo Boost* para el algoritmo que posee dos implementaciones, en teoría su tiempo de ejecución debería ser aún mayor que el observado actualmente.

4.3 Comparativa entre Docker y la Máquina 1.

La comparativa se basa en el rendimiento mostrado por la máquina 1 que representa a un S.O. y Docker en las métricas de tiempo de ejecución y consumo energético del paquete que obtuvieron los algoritmos estudiados. En lo que respecta a los algoritmos de ordenamiento, Docker requiere una mayor cantidad de tiempo de ejecución que la máquina 1 sin excepción alguna. Por el ámbito energético Docker presenta una ventaja en general para los 2 o 3 primeros tamaños de inputs de las categorías 1, 2 y 3, debido a que necesita una menor cantidad de energía. Esto se replica para el algoritmo Merge Sort Iterativo en la categoría 4. Sin embargo, para el resto de los tamaños de inputs la máquina 1 es más eficiente. Con relación a lo que incluye a la categoría 4, la máquina 1 requiere un menor consumo energético que su par para todos los tamaños de inputs, siendo la excepción la mencionada anteriormente. Cabe añadir que Merge Sort Recursivo no se adhiere a los comportamientos descritos, dado que Docker posee una menor eficiencia en tiempo de ejecución y consumo energético para todas las categorías y cantidades de elementos a buscar. En relación con los algoritmos de búsqueda, Binary Search Recursivo presenta un comportamiento distinto que los algoritmos de ordenamiento en relación con el tiempo. Esto debido a que Docker presenta una mayor eficiencia que la máquina 1 para todas las categorías y cantidad de elementos a buscar. En términos de eficiencia energética, cada algoritmo presenta distintos comportamientos, comenzando con Binary Search Iterativo, para el cual las 3 primeras categorías son dominadas en eficiencia por Docker. En cambio, la cuarta categoría se intercambia el dominio hacia la máquina 1. Otro comportamiento es el que presenta Binary Search Recursivo, el cual tiene un desempeño similar a la tendencia provista por los algoritmos de ordenamiento. Es decir, Docker obtiene un mejor desempeño en las primeras 3 categorías pero únicamente para los tamaños de inputs iniciales, dado que el resto son más eficientes en la máquina 1 al igual que toda la categoría 4. Por último, el algoritmo restante, es decir, Binary Search Tree no se pudo analizar por lo anteriormente explicado.

Dado todo lo anteriormente descrito, desde un punto de vista en torno a la eficiencia energética y tiempo de ejecución, es posible afirmar que Docker no produce un nivel de ventaja sobre el S.O. que sugiera optar por él. Esto se debe a que en la mayoría de los algoritmos, la máquina 1 es más rápida en ejecución, habiendo solo una excepción. Por el lado energético, existen ocasiones en las cuales Docker obtiene una ventaja sobre el S.O, pero son casos particulares que en grandes volúmenes de datos no suele ocurrir. Sin

embargo, en el caso específico de requerir utilizar alguno de los algoritmos estudiados para tamaños de archivos como los primeros dos a tres de las categorías 1, 2 y 3, podría ser una buena opción utilizar Docker si se busca solo eficiencia energética.

Capítulo 5. Conclusiones y Recomendaciones

A través del presente estudio se lograron capturar distintos comportamientos y tendencias de los algoritmos de ordenamiento y búsqueda con relación a las métricas de tiempo de ejecución y eficiencia energética. La principal síntesis que se obtuvo es que el tiempo de ejecución de un algoritmo no siempre es directamente proporcional con su consumo energético dentro de un mismo dispositivo. Esto queda demostrado en variadas situaciones en las cuales la ejecución de un algoritmo en una de las posibles máquinas o contenedor generaba el menor tiempo. Sin embargo, no era el más eficiente energéticamente al ser superado por uno de los dispositivos restantes. Otro punto es el hecho de que un algoritmo puede tener distintas tendencias en base al input con el cual se está ejecutando. Por otro lado, los algoritmos de ordenamiento al ser comparados con sus respectivas ejecuciones más eficientes muestran el mismo comportamiento en ambas métricas. En cambio, los algoritmos de búsqueda presentan distintas tendencias en base a la categoría en la cual deban realizar las búsquedas. Por ende, es posible inferir que los elementos y la cantidad de ellos en los cuales debe realizar las búsquedas son un factor primordial en relación con las métricas de tiempo y consumo energético. Cabe añadir, que en términos generales, las implementaciones iterativas presentan un mejor desempeño en lo que respecta a eficiencia, tanto de tiempo de ejecución como consumo energético del paquete si se compara la máquina más eficiente de cada implementación. Aunque para un mismo algoritmo la máquina más eficiente entre ambas implementaciones puede variar, los valores como tal benefician a la implementación iterativa al ser menores. Otro punto por mencionar es el hecho de que el tiempo y energía presenta una alta correlación con las métricas restantes, en especial con la cantidad de instrucciones, ciclos de CPU y fallas de caché, sin embargo, hubo situaciones donde su comportamiento no se veía reflejado en ellas. Esto permite teorizar el hecho de que pueden existir métricas que no fueron incluidas en el estudio y que poseen relevancia o algún tipo de factor no estudiado, como podría ser el desempeño de la arquitectura de los dispositivos. Por otro lado, la herramienta principal utilizada para llevar a cabo el estudio presentó limitaciones, principalmente la falta de capacidad de capturar valores generados por algunas métricas entre las categorías 1 y 3. A partir de aquello, hubo situaciones en las cuales no fue posible analizar a detalle todas las métricas, por ende, es recomendable utilizar una herramienta que asegure la captura de todos los valores. Sin embargo, esto no desmerece el beneficio que otorga *Perf* al ser una de uso gratuito.

Recomendaciones

Una de las primeras consideraciones que se debería tener en próximos estudios es el hecho de desactivar el *Intel Turbo Boost*, debido a que es una funcionalidad que no es controlable en el ambiente de experimentación. También, procurar que el estudio de las métricas sea acompañado de un análisis más a fondo de las arquitecturas de los computadores en los cuales se realicen los experimentos. Esto permitiría añadir más causales que permitan o ayuden a explicar los diferentes comportamientos vistos en los algoritmos y que no se reflejan en las métricas.

Trabajo futuro

El trabajo realizado podría ampliarse y seguir creciendo si se pueden realizar más experimentos, los cuales consideren modificar los archivos de entrada y máquinas a utilizar. Otro añadido es el poder agregar una variedad más grande de algoritmos dentro de los algoritmos de ordenamiento y búsqueda.

Dado que dentro de las conclusiones se obtuvo que el tiempo de ejecución no es directamente proporcional con la energía requerida, es un punto de inicio que podría ser estudiado más en profundidad. Esto permitiría llegar a un consenso que determine los algoritmos óptimos energéticamente en base a la funcionalidad que deberá cumplir y el hardware en el cual se vaya a ejecutar.

Los códigos de los algoritmos se encuentran disponibles en el siguiente repositorio de Github que es de libre acceso https://github.com/ldelafuente99/EnergyConsumption. Por otro lado, los inputs utilizados y datos crudos obtenidos a partir de la experimentación realizada se almacenaron en Google Drive en la siguiente URL https://cutt.ly/fnUyhgq, compartiendo la característica que son de libre acceso.

Referencias

1. Pang Candy, Hindle Abram, Adams Bram, Hassan Ahmed E, (2015). What do programmers know about the energy consumption of software?. In IEEE Software.
2. Shaw Keith, (2019). What is edge computing and why it matters. Accedido el 27 de enero, 2021, desde https://tinyurl.com/y2zxfx5c.
3. Strubell Emma, Ganesh Ananya, McCallum Andrew, (2019). Energy and Policy Considerations for Deep Learning in NLP. College of Information and Computer Sciences University of Massachusetts Amherst.
4. Gurin Sebastián, (2004). Algoritmos de ordenación. Free Software Foundation.
5. Algoritmos de ordenamiento. Accedido el 27 de enero, 2021 desde https://tinyurl.com/y2vs92wc.
6. Knuth, Donald (1998). “Section 5.2.4: Sorting by Merging”. Sorting and Searching. The Art of Computer Programming, 3 (2nd ed.). Addison-Wesley. pp. 158-168.
7. Wachenchauzer Rosita, Manterola Margarita, Curia Maximiliano, Medrano Marcos, Paez Nicolás, (2014). Algoritmos de Programación con Python. Accedido el 28 de enero, 2021 desde https://tinyurl.com/y58g7pad.
8. Algoritmo MergeSort, Curso de Programación II, Universidad Nacional del Oeste. Accedido el 28 de enero, 2021, desde https://tinyurl.com/yyks7j27.
9. Cormen Thomas H., Leiserson Charles E., Rivest Ronald L., Stein Clifford, (2001). Introduction to Algorithms, 2nd Edition. MIT Press and McGraw-Hill. Section 8.2: Counting sort, pp. 168-170.
10. Knuth Donald (1998). The Art of Computer Programming, Volume 3: Sorting and Searching. 2nd Edition. Addison-Wesley. Section 5.2: Sorting by counting, pp.75-80.
11. Programiz. Counting Sort Algorithm. Accedido el 4 de febrero, 2021, desde https://tinyurl.com/y9sfyq23.
12. Williams J. W. J., (1964). “Algorithm 232-Heapsort”. Communications of the ACM, pp.347-348.
13. Pier Paolo Guillen Hernandéz, (2011). Ordenamiento por montículos (*Heapsort*). Accedido el 4 de febrero, 2021, desde https://tinyurl.com/y7by26wp.
14. EcuRed, (2012). Algoritmo de búsqueda. Accedido el 5 de febrero, 2021, desde https://tinyurl.com/ybxlr2cb.
15. Cormen Thomas, Balkcom Devin, (2015). Búsqueda binaria. Accedido el 6 de febrero, 2021, desde https://tinyurl.com/yckhaky4.

16. Williams Jr., Louis F, (1975). A modification to the hald-interval search (binary search) method Proceedings of the 14th ACM Southeast Conference, pp.95-101.
17. TutorialesPoint, (2021). Data Structure – Binary Search Tree. Accedido el 8 de febrero, 2021, desde https://tinyurl.com/y7mdyvax.
18. Exposito López Daniel, Soto García Abraham, Martín Gómez Antonio José, (2000). Árboles Binarios de Búsqueda. Accedido el 8 de febrero, 2021, desde https://tinyurl.com/y9cm9n45.
19. GeekforGeeks, (2020). Binary Heap. Accedido el 9 de febrero, 2021, desde https://tinyurl.com/y7aeh3an.
20. Raffino Maria Estela, (2020). Procesador. Accedido el 10 de junio, 2021, desde https://concepto.de/procesador/.
21. Isidro Ros, (2019). Memoria caché: qué es y qué diferencias hay entre los tipos L1, L2 y L3. Accedido el 10 de febrero, 2021, desde https://tinyurl.com/ybv5rvlm.
22. Significado de Memoria caché. Accedido el 10 de febrero, 2021, desde https://tinyurl.com/y88jobkq.
23. Significados, (2017). Qué es la memoria caché L1, L2 y L3 y cómo funciona. Accedido el 10 de febrero, 2021, desde https://tinyurl.com/y9wxeryy.
24. Landers Jhon, (2021). ¿Qué es la memoria principal de una computadora?. Accedido el 12 de febrero, 2021, desde https://tinyurl.com/y75bmg6h.
25. Alloza Martín Jesús, (2014). Montaje de componentes y periféricos microinformáticos. IFCT0108. Google libros.
26. Weaver M Vincent, (2013). Linux perf_event Features and Overhead. FastPath Workshop.
27. Khan Kashif Nizam, Hirki Mikael, Niemi Tapio, Nurminen Jukka K., Ou Zhonghong, (2018). RALP in Action: Experiences in Using RAPL for Power Measurements. ACM Trans. Model. Perform. Eval. Comput. Syst. 3.2, Article 9 (March 2018), 26 pages.
28. Mendoza Marvin, (2019). Qué es un lenguaje de programación. Lenguajes de programación. Accedido el 13 de febrero, 2021, desde https://tinyurl.com/yakh4dd7.
29. Pereira Rui, Couto Marco, Ribeiro Francisco, Rua Rui, Cunha Jácome, Fernandes J. P., Saraiva J., (2017). Energy Efficiency across Programming Languages: How Do Enery, Time, and Memory Relate?. In Proceedings of 2017 ACM SIGPLAN International Conference on Software Language Engineering (SLE' 17). ACM, New York, NY, USA, 12 pages.

30. Holloway Christopher, (2018). ¿Qué son los contenedores de software y cómo aportan valor a las organizaciones?. IT Master Mag. Accedido el 13 de febrero, 2021, desde https://tinyurl.com/yct7mof7.
31. O'Gara, Maureen, (2013). Ben Golub, Who Sold Gluster to Red hat, Now Running dotClouds. SYS-CON Media.
32. Pierre Carbonnelle, (2020). PYPL PopularitY of Programming Language. Accedido el 13 de marzo, 2021, desde https://pypl.github.io/PYPL.html.
33. José Antonio Castillo, (2019). Partes de un procesador fuera y dentro: conceptos básicos. Accedido el 19 de marzo, 2021, desde https://tinyurl.com/yzwezg4m.
34. Perf Wiki. *perf: Linux Profiling with Perfomarnce Counters*. Accedido el 16 de marzo, 2021, desde https://cutt.ly/7nUf6DE.
35. Naif Aljabri, Muhammed Al-Hashimi, Mostafa Saleh, Osama Abulnaja, (2019). Investigating power efficiency of mergesort. The Journal of Supercomputing.
36. GeeksforGeeks, (2020). Merge Sort. Accedido el 6 de noviembre, 2020, desde https://cutt.ly/jnUf7Zh.
37. Techie Delight, (2020). Iterative Merge Sort Algorithm (Bottom-up Merge Sort). Accedido el 13 de noviembre de 2020, desde https://cutt.ly/BnUf8ax.
38. John Papiewski, (2018). ¿Que es la jerarquía de almacenamiento?. Accedido el 9 de junio, 2021, desde https://cutt.ly/QnTeMqi.

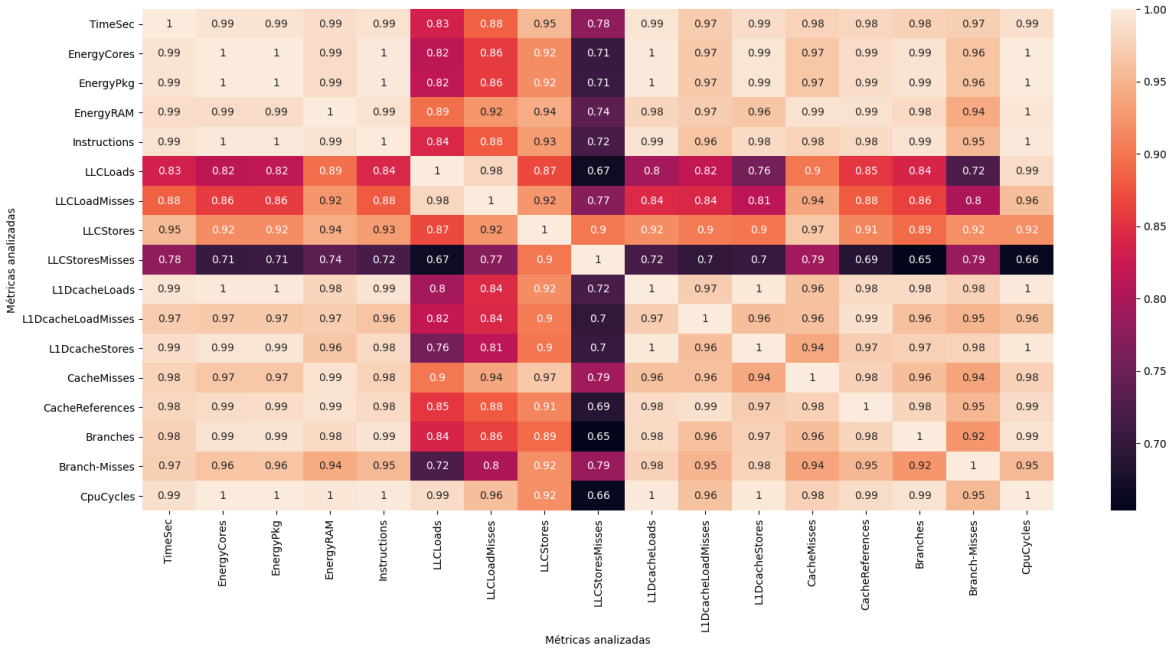
Referencias Imágenes

39. Wikipedia, (2021). Stable sorting algorithm. Accedido el 19 de enero de 2021, desde https://simple.wikipedia.org/wiki/Sorting_algorithm.
40. Enrique García Hernández, (2015). Métodos de ordenación MergeSort. Accedido el 26 de enero, 2021, desde https://cutt.ly/8ngaeEE.
41. Lucas Machado, Bruno Feijó, (2009). Parallel culling and sorting based on adaptive static balancing. Accedido el 28 de enero, 2021, desde https://cutt.ly/xngakch.
42. GeekforGeeks, (2021). HeapSort. Accedido el 28 de febrero, 2021, desde https://cutt.ly/lngdYwi.
43. Parse Objects, (2018). Binary Search Algorithm in JavaScript. Accedido el 9 de marzo, 2021, desde https://cutt.ly/VngfKSX.
44. Wikipedia, (2021). Binary search tree. Accedido el 9 de marzo, 2021, desde https://cutt.ly/AngfVUO.
45. José Antonio Castillo, (2019). Qué es la memoria caché L1, L2 y L3 y cómo funciona. Accedido el 11 de marzo, 2021, desde https://cutt.ly/2ngf2oX.
46. Hugo de Juan, (2016). Introducción a Docker. Accedido el 21 de marzo, 2021, desde https://cutt.ly/Angf6Sq.

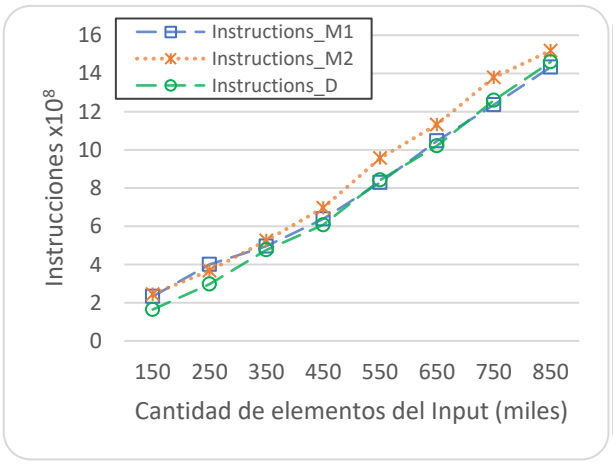
Anexos

Anexo 1.

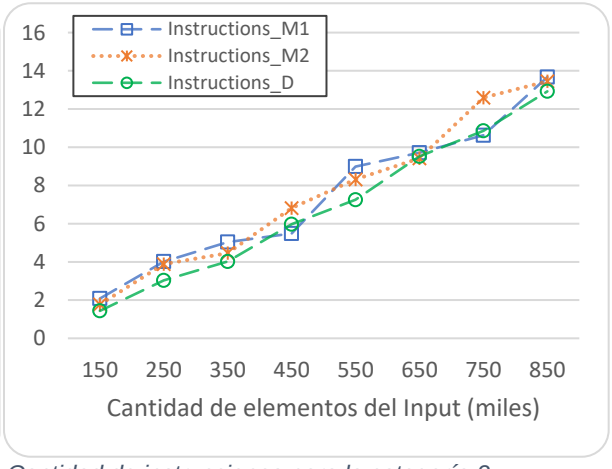
- Matriz de correlación para la máquina 2 en el Merge Sort Iterativo.



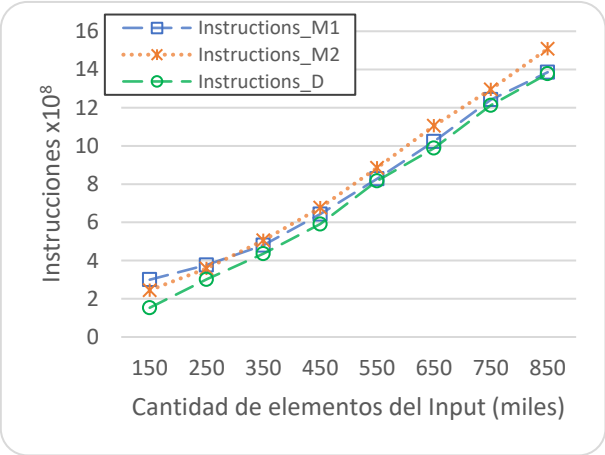
- Cantidad de instrucciones, LLCStoresMisses y Branches de las categorías 1, 2 y 3 para la máquina 2 en el Merge Sort Iterativo.



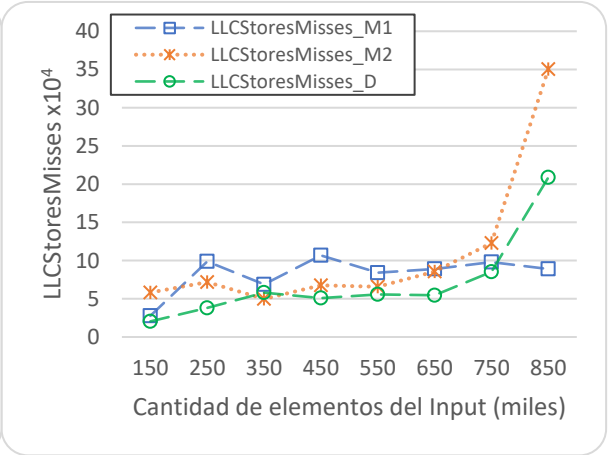
Cantidad de instrucciones para la categoría 1.



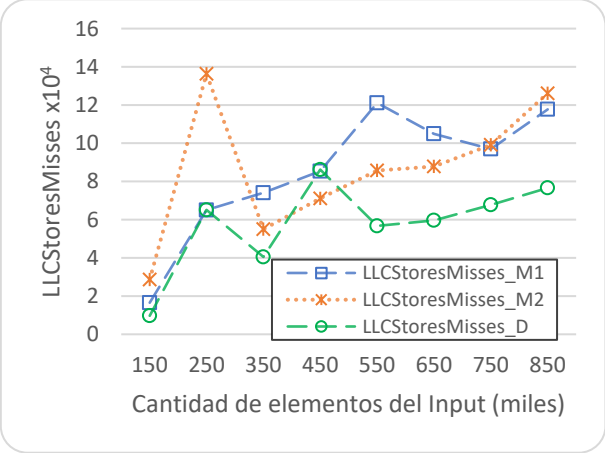
Cantidad de instrucciones para la categoría 2.



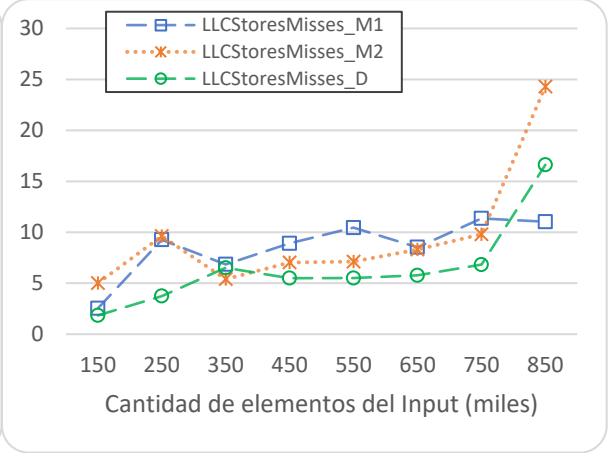
Cantidad de instrucciones para la categoría 3.



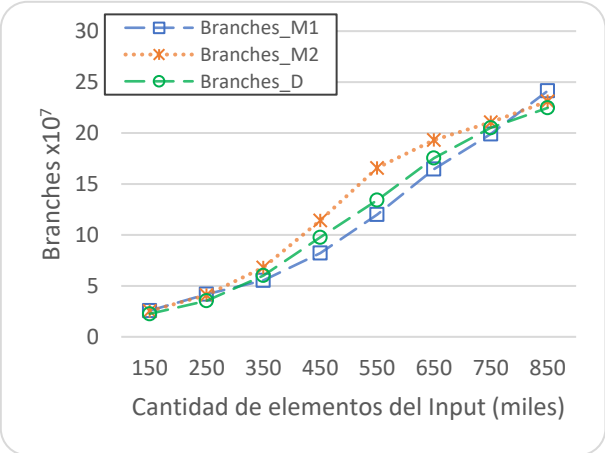
LLCStoresMisses para la categoría 1.



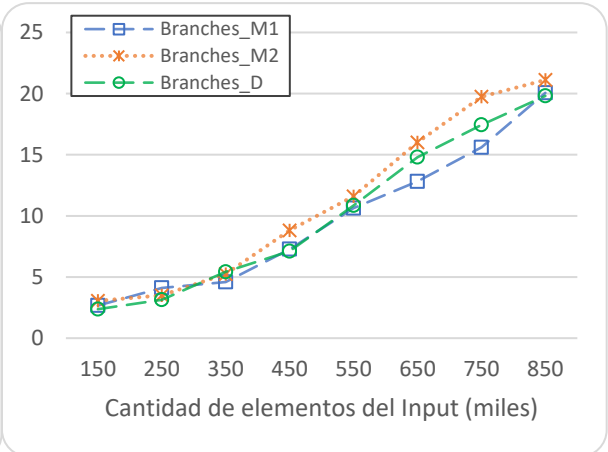
LLCStoresMisses para la categoría 2.



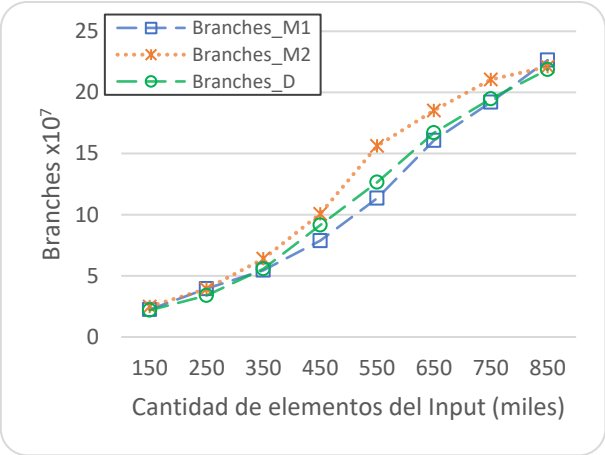
LLCStoresMisses para la categoría 3.



Branches para la categoría 1.



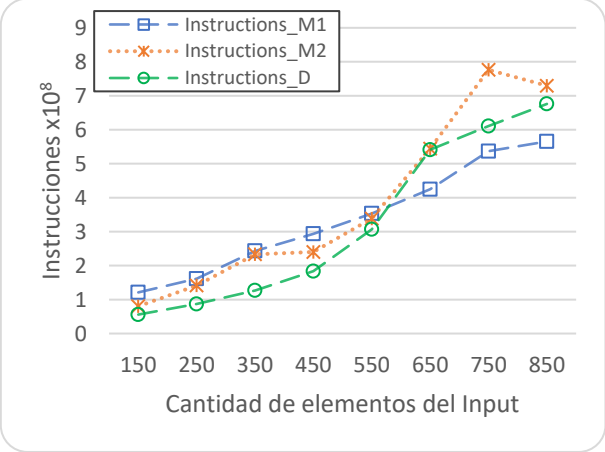
Branches para la categoría 2.



Branches para la categoría 3.

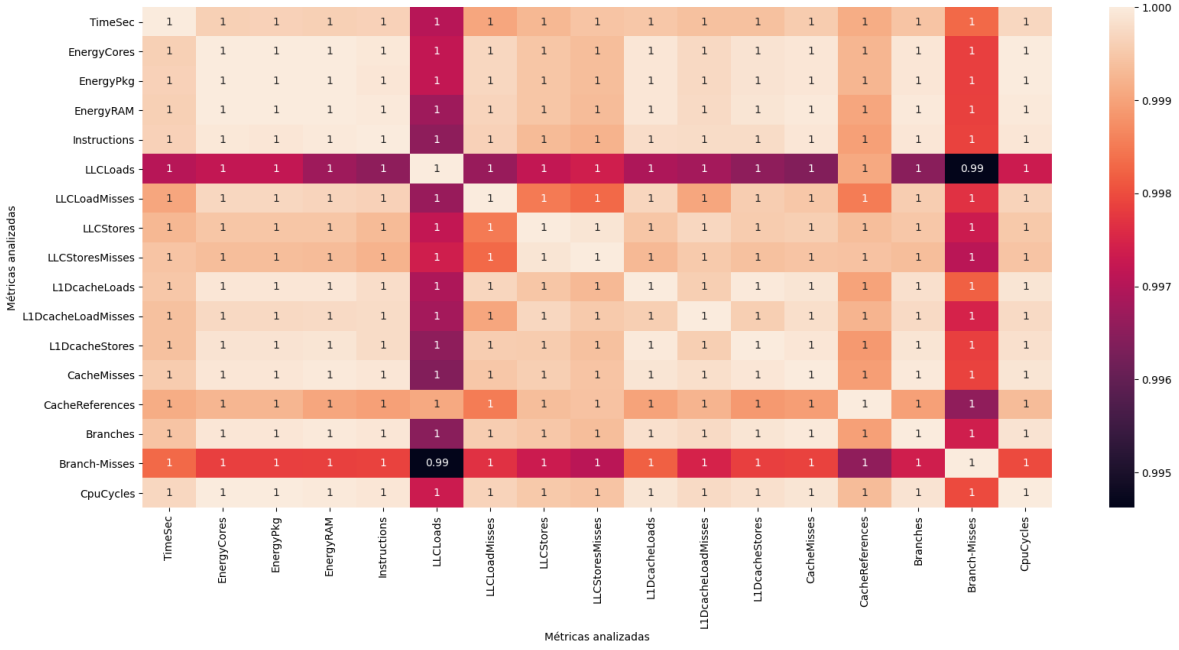
Anexo 2

- Cantidad de instrucciones Counting Sort.



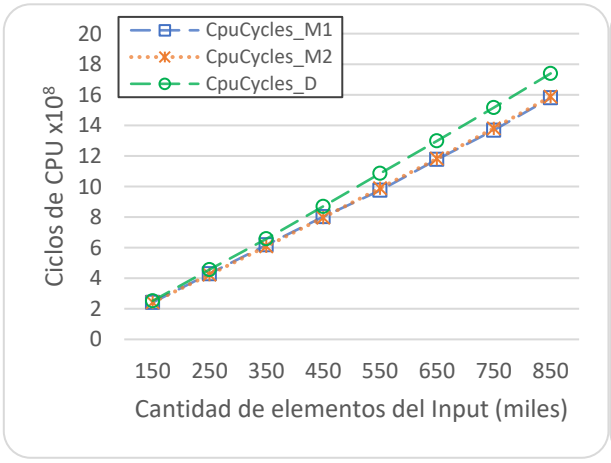
Categoría 1.

- Matriz de correlación para la categoría 4 de Counting Sort.

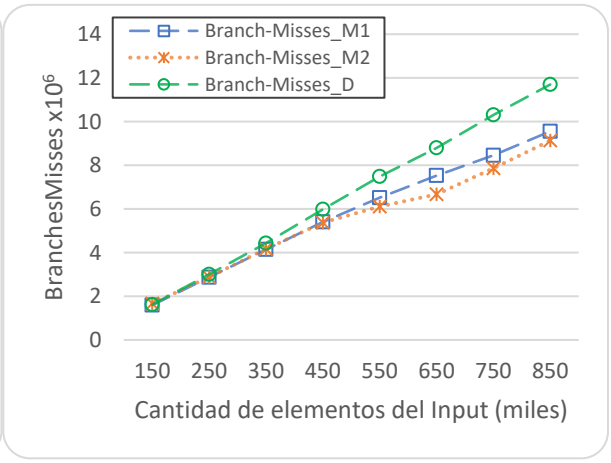


Anexo 3

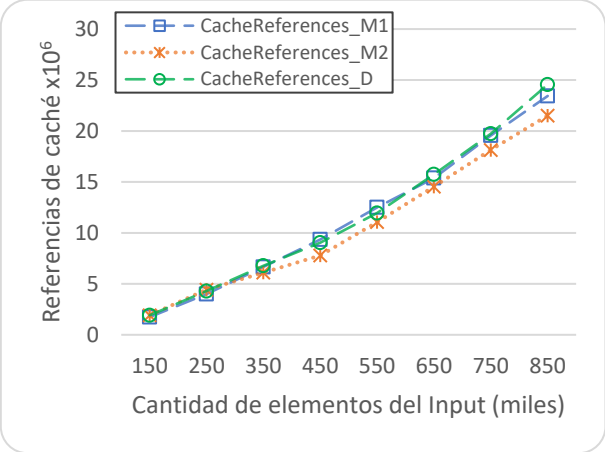
- Ciclos de CPU, BranchesMisses y referencias de caché para la categoría 1 de Heap Sort.



Ciclos de CPU.



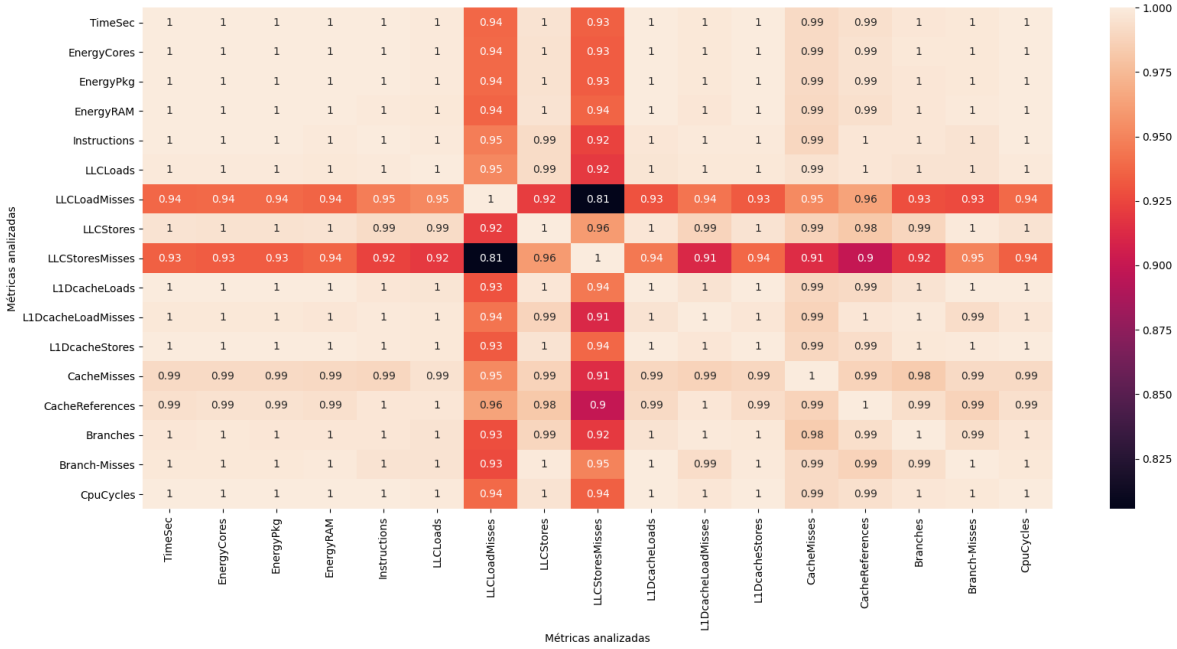
BranchesMisses.



Referencias de caché.

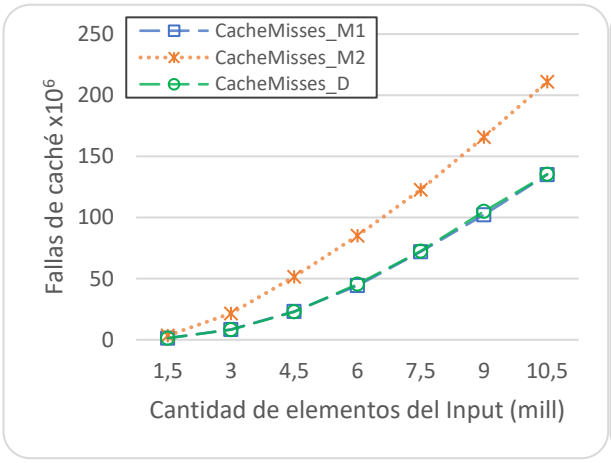
Anexo 4

- Matriz de correlación para la categoría 1 de Heap Sort.

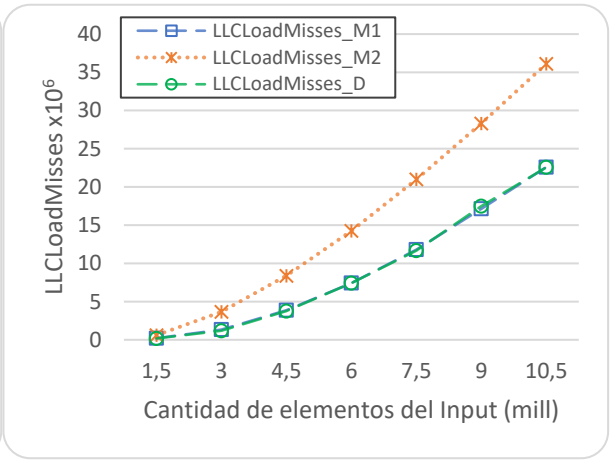


Anexo 5

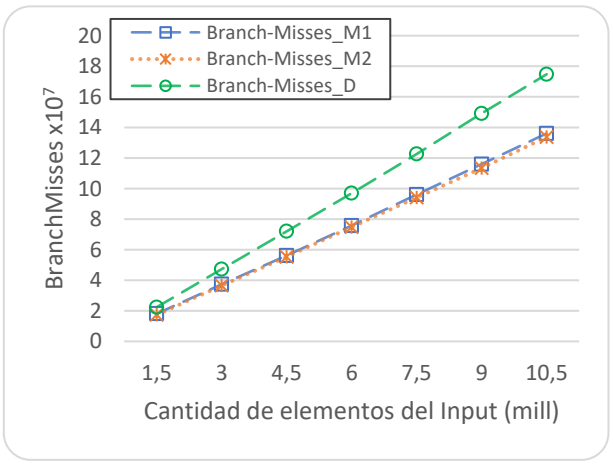
- Fallas de caché, LLCLoadMisses y BranchMisses para la categoría 4 de Heap Sort.



Fallas de caché.



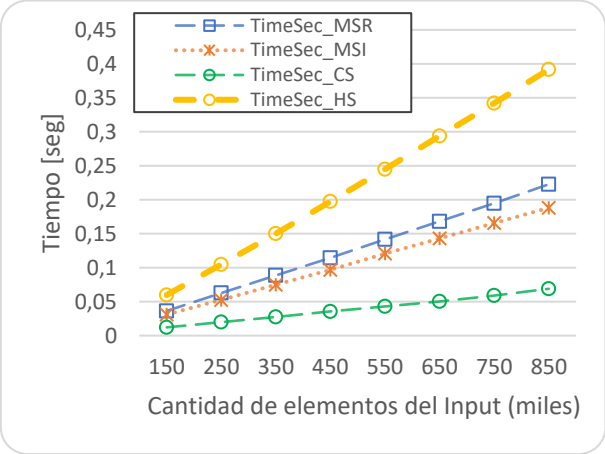
LLCLoadMisses



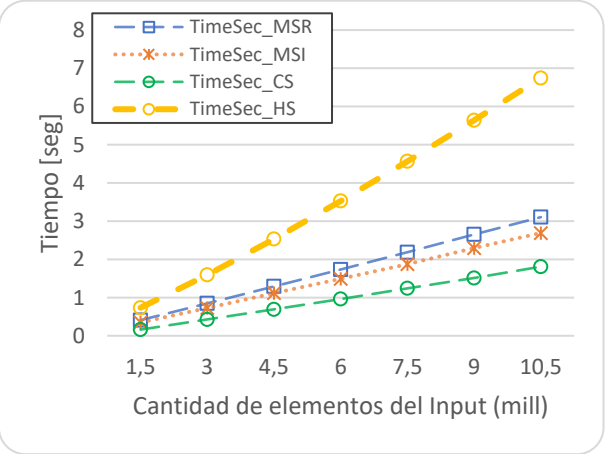
BranchMisses.

Anexo 6

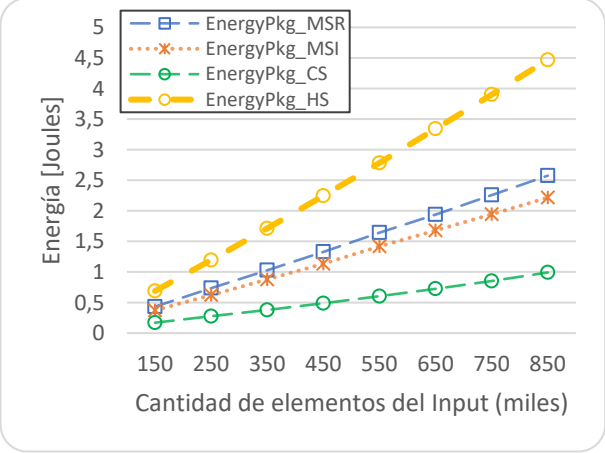
- Tiempo de ejecución de la máquina 1 y energía del paquete de la máquina 2 utilizada por los algoritmos de ordenamiento para las categorías 1 y 4.



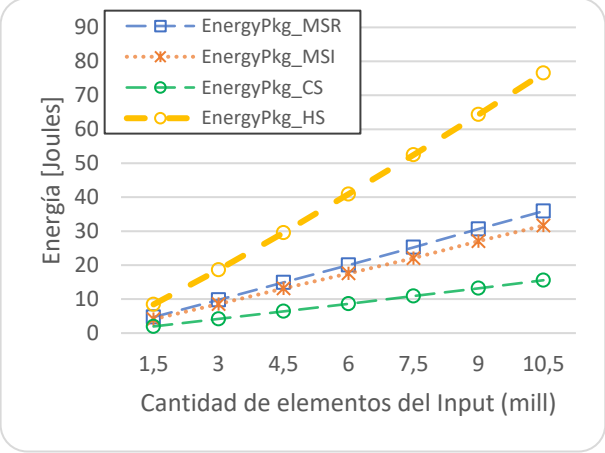
Tiempo de ejecución de la categoría 1.



Tiempo de ejecución de la categoría 4.



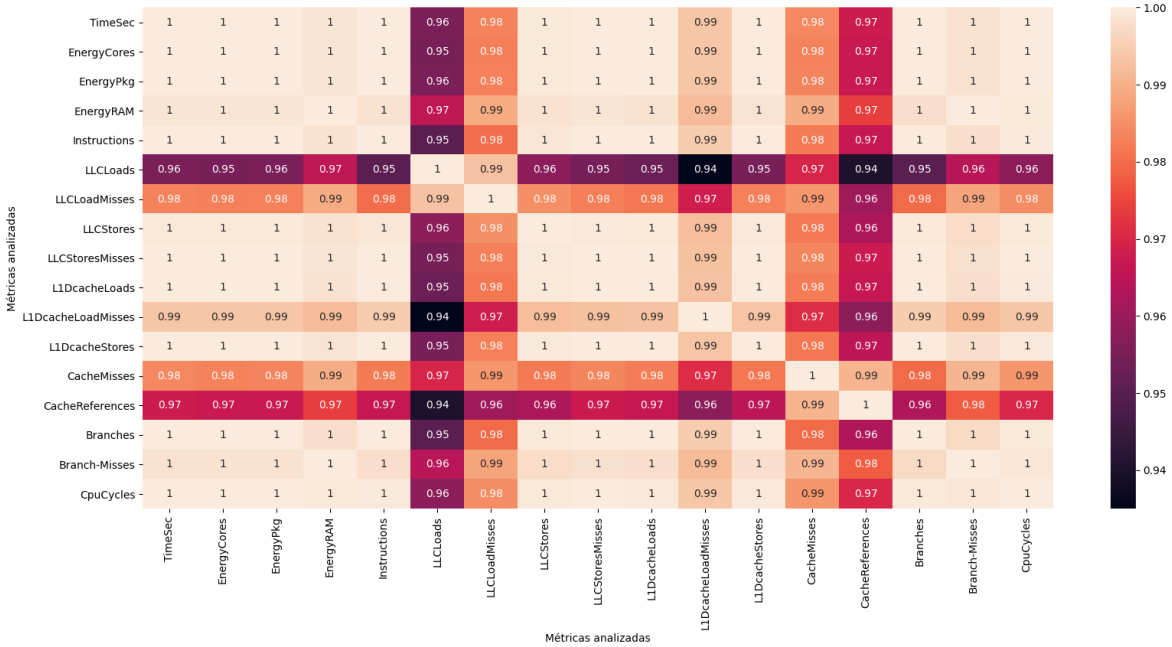
Energía del paquete para la categoría 1.



Energía del paquete para la categoría 4.

Anexo 7

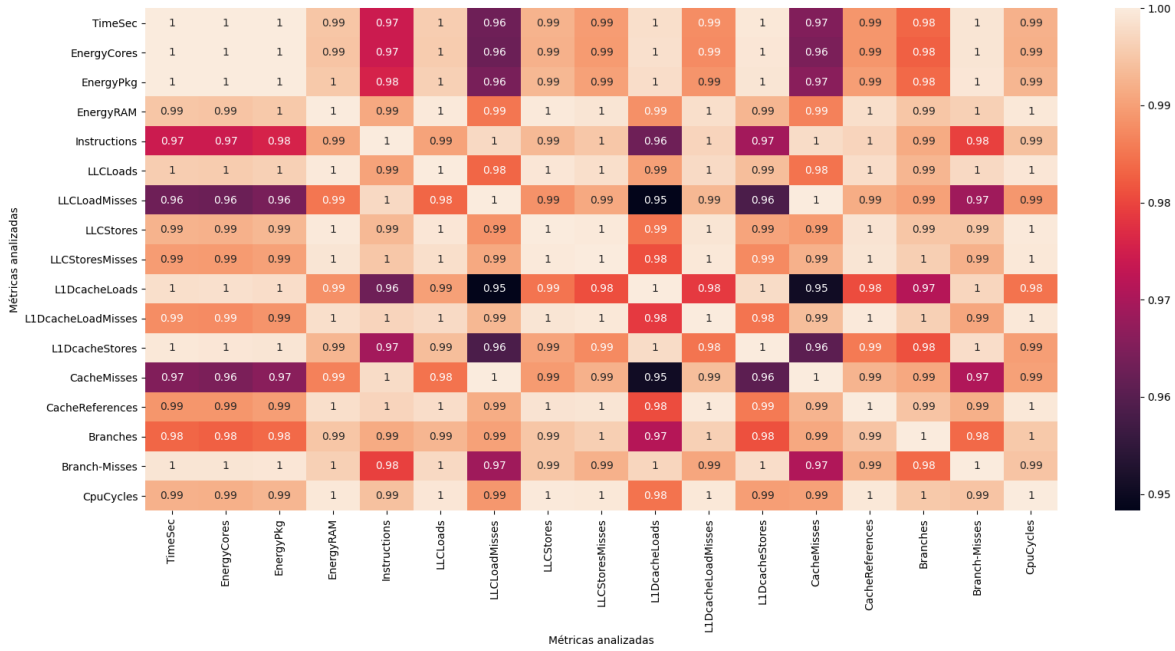
- Matriz de correlación de Docker para la categoría 4 con 150.000 búsquedas en Binary Search Iterativo.



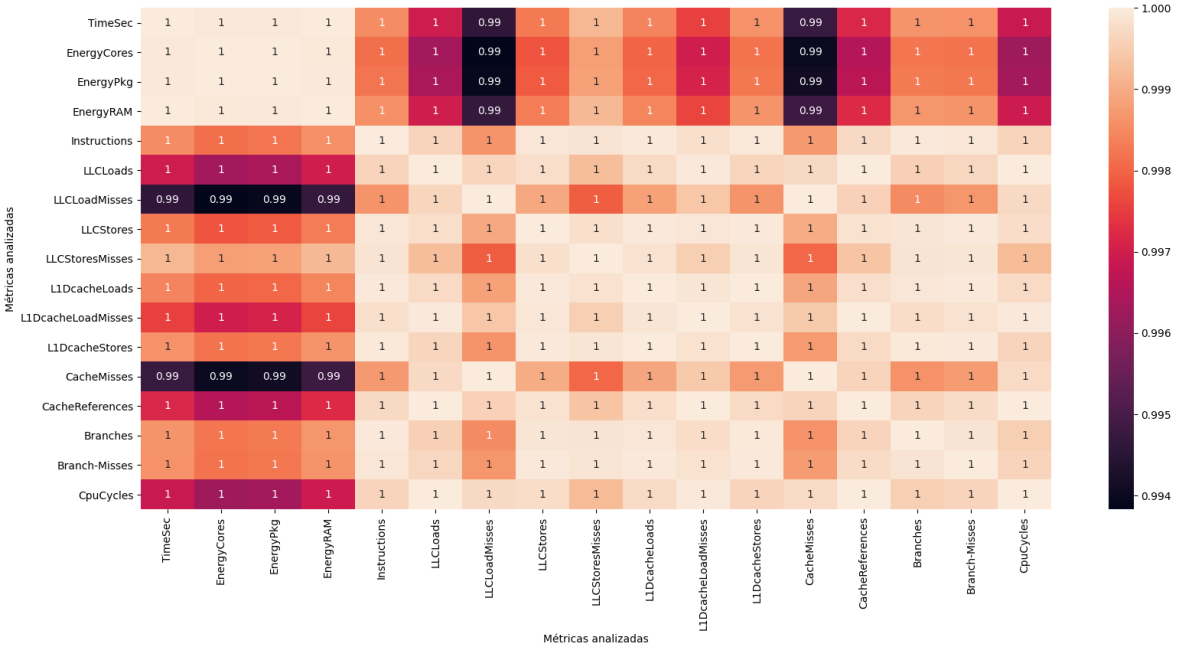
Matriz de correlación de la categoría 4 para 150.000 búsquedas en Docker

Anexo 8

- Matriz de correlación de Binary Search Tree para las categorías 1 y 4 con 150.000 búsquedas.
-



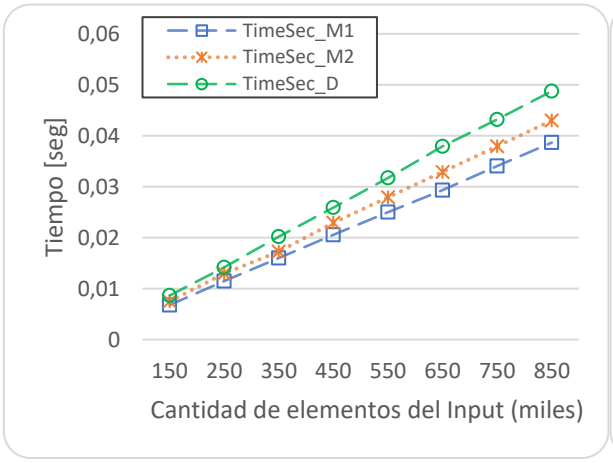
Matriz de correlación del algoritmo Binary Search Tree para la categoría 1 con 150.000 búsquedas.



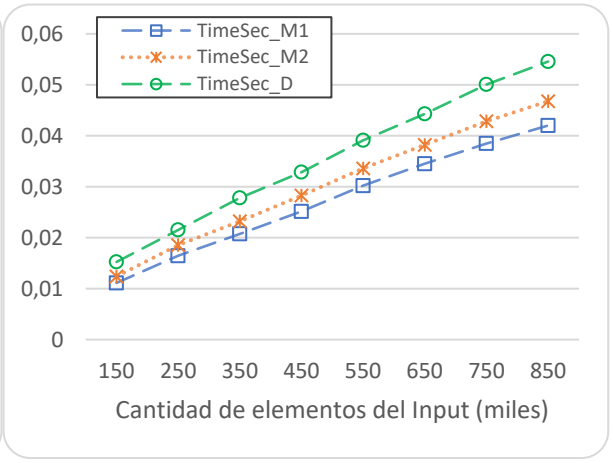
Matriz de correlación del algoritmo Binary Search Tree para la categoría 4 con 150.000 búsquedas.

Anexo 9

- Tiempo de ejecución de la categoría 2 del algoritmo Binary Search Tree con 0 y 150.000 búsquedas.



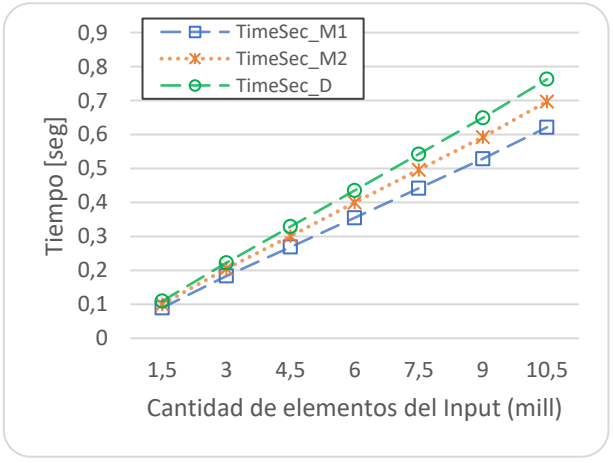
Tiempo de ejecución de la C. 2 con 0 búsquedas.



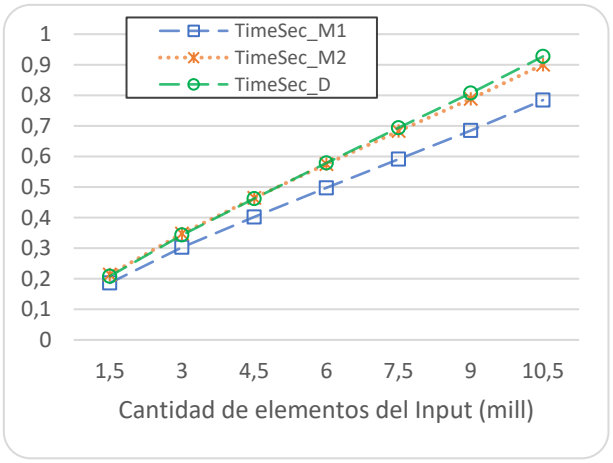
Tiempo de ejecución de la C. 2 con 150.000 búsquedas.

Anexo 10

- Tiempo de ejecución de la categoría 4 del algoritmo Binary Search Tree con 0 y 150.000 búsquedas.



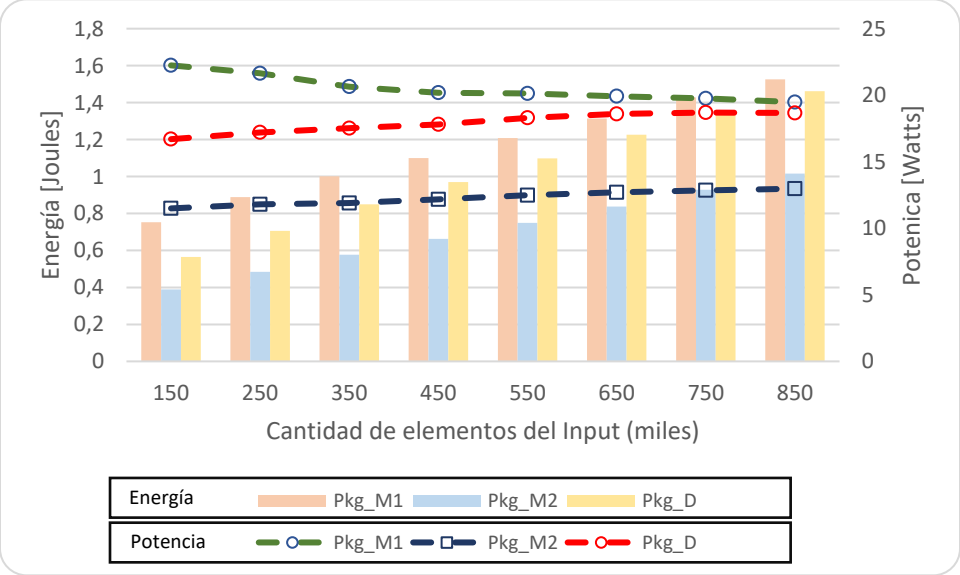
Tiempo de ejecución de la C. 4 con 0 búsquedas.



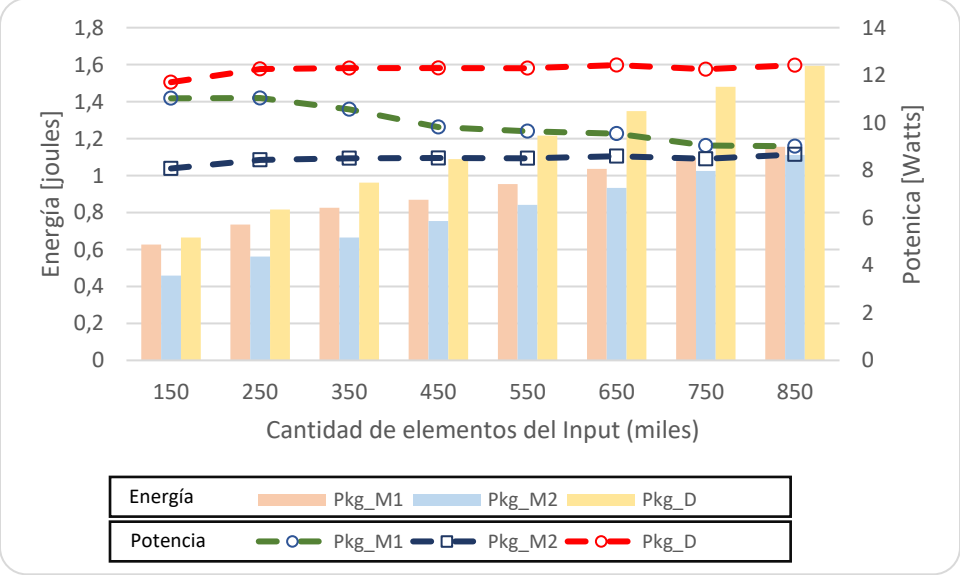
Tiempo de ejecución de la C. 4 con 150.000 búsquedas.

Anexo 11

- Consumo energético del paquete y potencia de la categoría 1 con 150.000 búsquedas para Binary Search Iterativo y Recursivo.



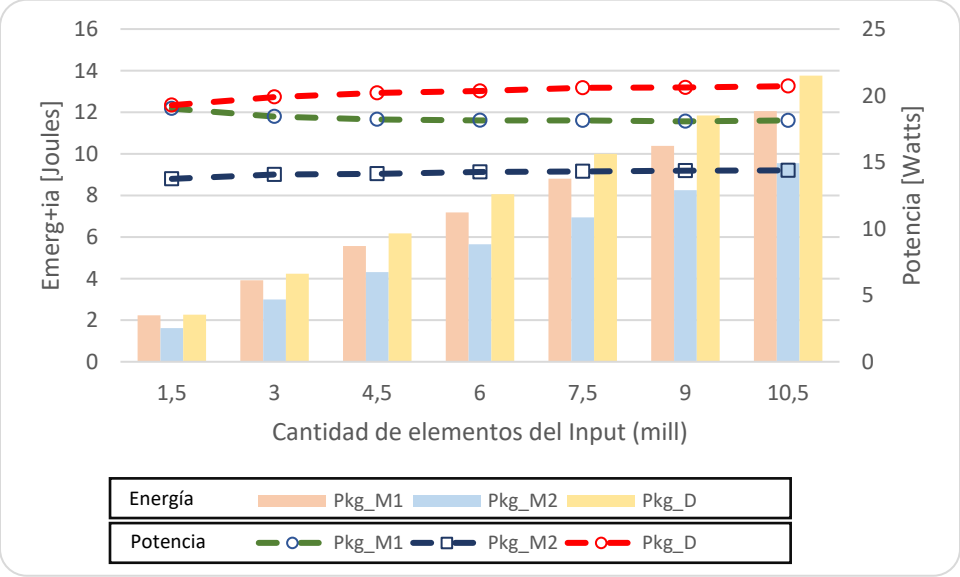
Consumo energético del paquete y potencia del algoritmo Binary Search Iterativo con 150.000 búsquedas en la C.1.



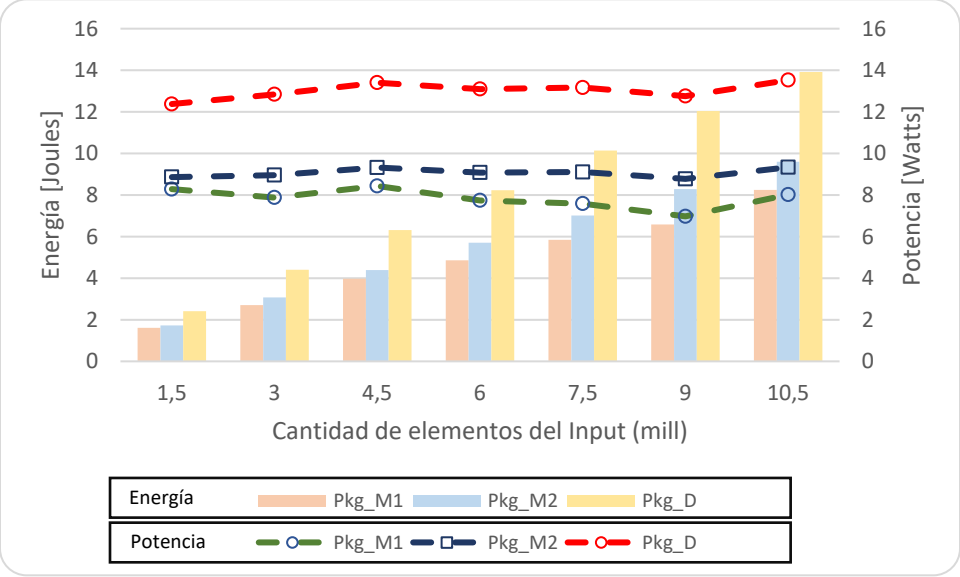
Consumo energético del paquete y potencia del algoritmo Binary Search Recursivo con 150.000 búsquedas en la C.1.

Anexo 12

- Consumo energético del paquete y potencia de la categoría 4 con 150.000 búsquedas para Binary Search Iterativo y Recursivo.



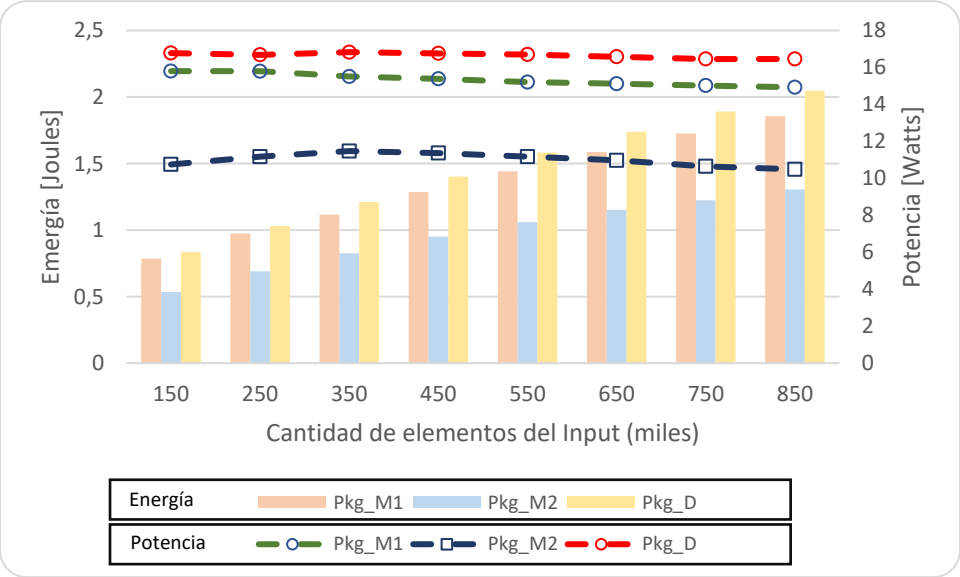
Consumo energético del paquete y potencia del algoritmo Binary Search Iterativo con 150.000 búsquedas en la C.4.



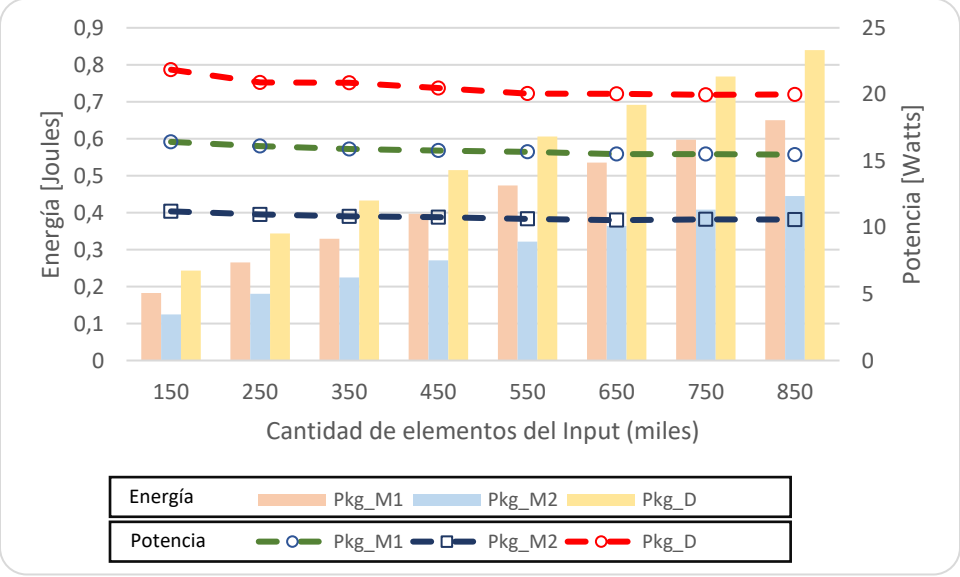
Consumo energético del paquete y potencia del algoritmo Binary Search Recursivo con 150.000 búsquedas en la C.4.

Anexos 13

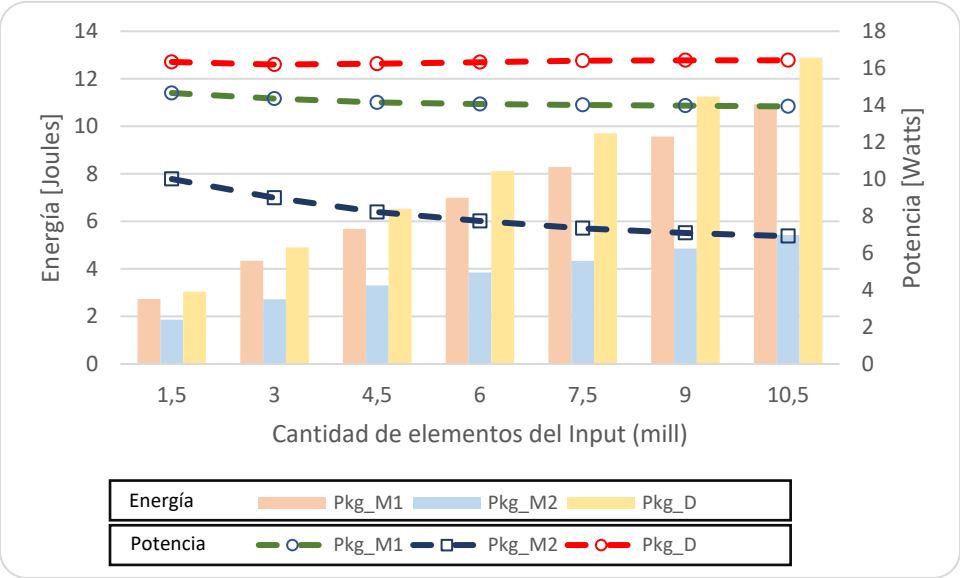
- Consumo energético del paquete y potencia del algoritmo Binary Search Tree con 150.000 búsquedas en las categorías 1, 2 y 4.



Consumo energético del paquete y potencia del algoritmo Binary Search Tree con 150.000 búsquedas en la C.1.



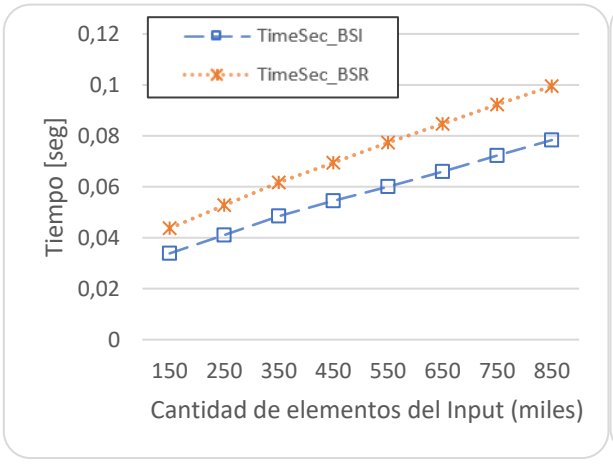
Consumo energético del paquete y potencia del algoritmo Binary Search Tree con 150.000 búsquedas en la C.2.



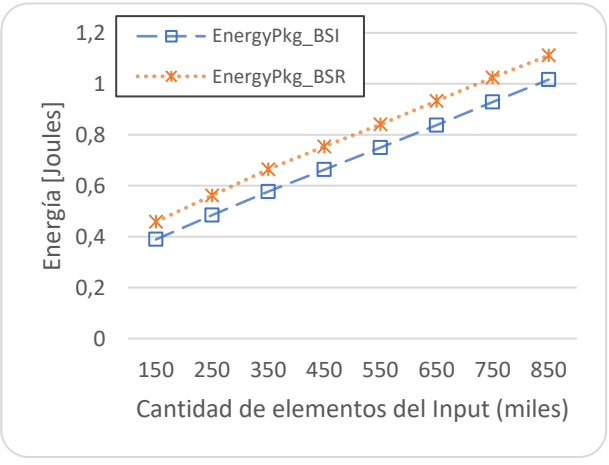
Consumo energético del paquete y potencia del algoritmo Binary Search Tree con 150.000 búsquedas en la C.4.

Anexo 14

- Tiempo de ejecución y consumo energético del paquete para los algoritmos de Binary Search con 150.000 búsquedas para la categoría 1.



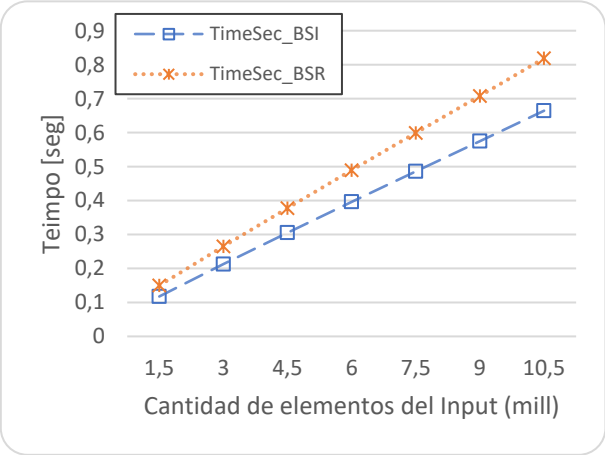
Tiempo de ejecución.



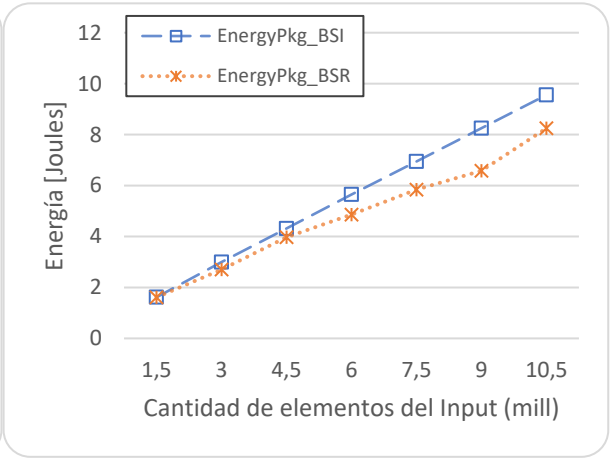
Consumo energético del paquete.

Anexo 15

- Tiempo de ejecución y consumo energético del paquete para los algoritmos de Binary Search con 150.000 búsquedas para la categoría 4.



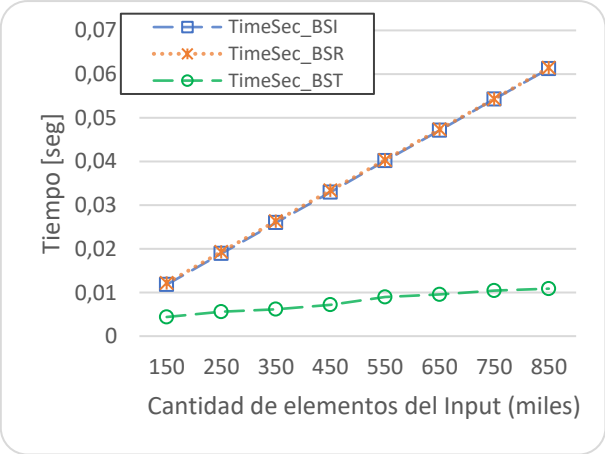
Tiempo de ejecución.



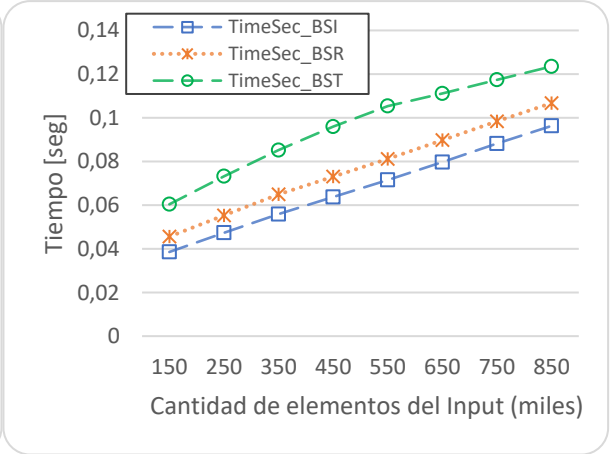
Consumo energético del paquete.

Anexo 16

- Tiempo de ejecución de la máquina 2 en los algoritmos de búsqueda con 10.000 y 150.000 búsquedas en la categoría 1.



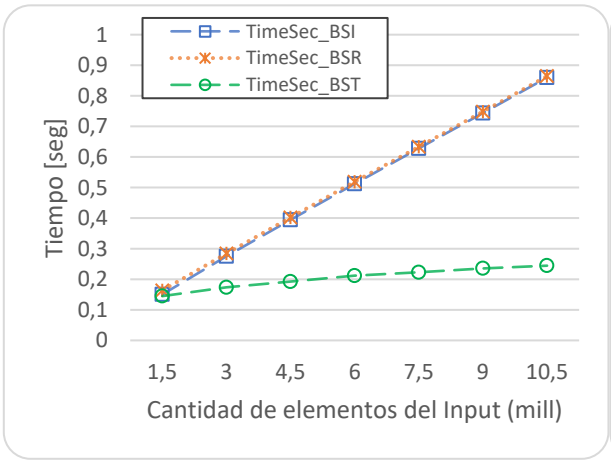
Tiempo de ejecución con 10.000 búsquedas.



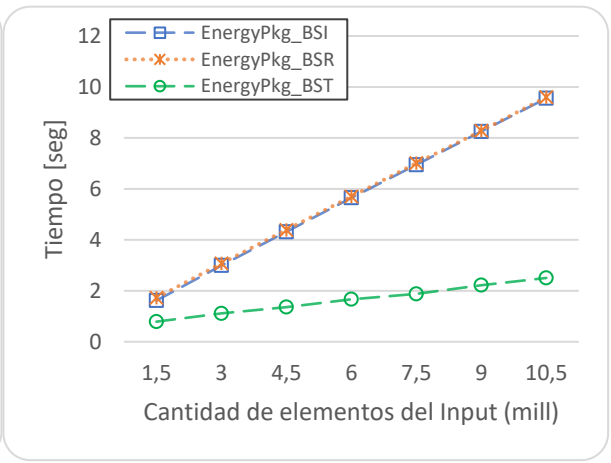
Tiempo de ejecución con 150.000 búsquedas.

Anexo 17

- Tiempo de ejecución y consumo energético del paquete de la máquina 2 en los algoritmos de búsqueda con 150.000 búsquedas en la categoría 4.



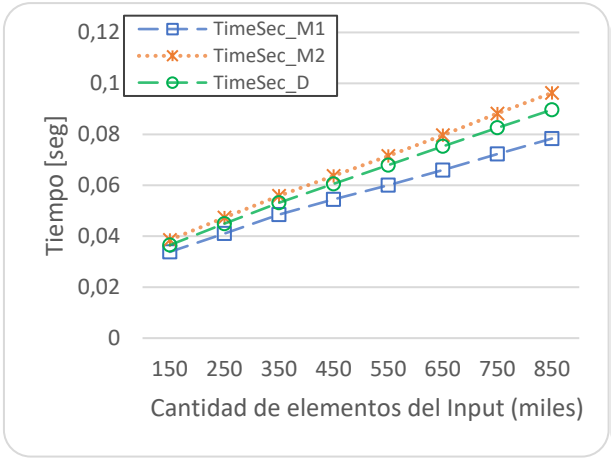
Tiempo de ejecución.



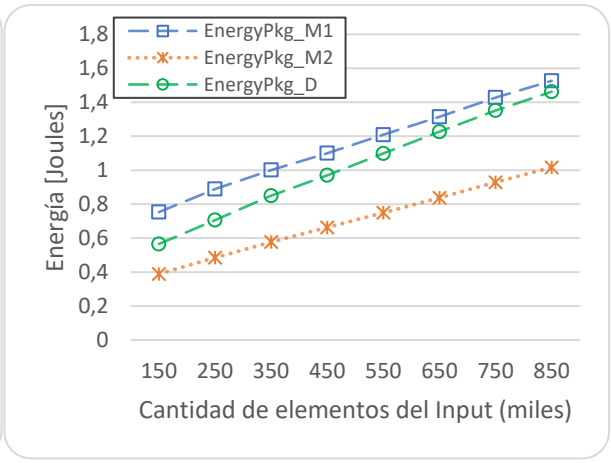
Consumo energético del paquete.

Anexo 18

- Tiempo de ejecución y consumo energético del paquete del algoritmo Binary Search Iterativo en la categoría 1 con 150.000 búsquedas.



Tiempo de ejecución.



Consumo energético del paquete.