UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

Graduado en Ingeniería Informática

Privatización parcial de Matrix Profile usando Memoria Transaccional

Partial privatization of Matrix Profile using Transactional Memory

Realizado por
Ignacio Paez Bertarini

Tutorizado por
Ricardo Quislant del Barrio
Eladio Gutiérrez Carrasco

Departamento
Arquitectura de Computadores

Málaga, septiembre de 2022

UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADUADO EN INGENIERÍA INFORMÁTICA

Privatización parcial de Matrix Profile usando Memoria Transaccional

Partial privatization of Matrix Profile using Transactional Memory

Realizado por
Ignacio Paez Bertarini

Tutorizado por
Ricardo Quislant del Barrio
Eladio Gutiérrez Carrasco

Departamento
Arquitectura de Computadores

UNIVERSIDAD DE MÁLAGA

MÁLAGA, SEPTIEMBRE DE 2022

Fecha defensa: OCTUBRE de 2022

Abstract

Matrix Profile is established as one of the methods for time series data mining. This algorithm solves the problem of finding motifs and anomalies in large time series accurately, quickly and without the need for heuristic parameterizations. Its parallelization is feasible by privatizing or replicating the data structures on which it is based and executing a final reduction stage, but this can be a problem when the series are very large.

On the other hand, the Transactional Memory (TM) paradigm seeks to simplify parallel programming and improve performance by introducing the concept of transaction. A transaction is a piece of code that is executed atomically and in isolation from other transactions as well as in parallel with them. The transactional system (software or hardware) is responsible for detecting dependencies between transactions and serializing the execution in case of conflict.

This TFG proposes the parallelization of the Matrix Profile algorithm using the Transactional Memory paradigm, with the intention of reducing the memory consumption of the algorithm without worsening its execution time.

Key words: **MatrixProfile, RTM, HTM**

Resumen

Matrix Profile está consolidado como uno de los métodos para la minería de datos en series temporales. Este algoritmo soluciona el problema de encontrar patrones y anomalías en series temporales de gran tamaño de manera exacta, rápida y sin la necesidad de parametrizaciones heurísticas. Su paralelización es factible privatizando o replicando las estructuras de datos en las que se basa y ejecutando una etapa de reducción final, pero esto puede resultar un problema cuando las series son muy grandes.

Por otro lado, el paradigma de Memoria Transaccional (TM) busca simplificar la programación paralela y mejorar el rendimiento introduciendo el concepto de transacción. Una transacción es un trozo de código que se ejecuta de manera atómica y aislada de otras transacciones a la vez que en paralelo a ellas. El sistema transaccional (software o hardware) se encarga de detectar dependencias entre transacciones y serializar la ejecución en caso de conflicto.

En este TFG se plantea la paralelización del algoritmo Matrix Profile usando el paradigma de Memoria Transaccional, con la intención de reducir el consumo de memoria del algoritmo sin empeorar el tiempo de ejecución del mismo.

Palabras clave: **MatrixProfile, RTM, HTM**

Índice

1 Introducción

1.1. Motivación

Matrix Profile está consolidado como uno de los métodos para la minería de datos en series temporales [9, 10, 11, 18, 20, 21, 22, 23, 24]. Este algoritmo soluciona el problema de encontrar patrones y anomalías en series temporales de gran tamaño de manera exacta, rápida y sin la necesidad de parametrizaciones heurísticas. Su paralelización es factible privatizando o replicando las estructuras de datos en las que se basa y ejecutando una etapa de reducción final, pero esto puede resultar un problema cuando las series son muy grandes (e.g. Una serie de 500 millones de elementos tipo double ocupa aproximadamente 4GB de memoria. A esto hay que añadirle varios vectores de estadísticos que pueden ocupar entre 8GB y 12GB, más las estructuras de datos necesarias para el Matrix Profile, que son dos vectores de aproximadamente 4GB y 2GB respectivamente).

Por otro lado, el paradigma de Memoria Transaccional (TM) busca simplificar la programación paralela y mejorar el rendimiento introduciendo el concepto de transacción. Una transacción es un trozo de código que se ejecuta de manera atómica y aislada de otras transacciones a la vez que en paralelo a ellas. El sistema transaccional (software o hardware) se encarga de detectar dependencias entre transacciones y serializar la ejecución en caso de conflicto. En el caso de Hardware TM (HTM) el uso frecuente y prolongado de transacciones puede llevar a una infrautilización de los recursos de la CPU debido a que se introducen constantemente barreras de memoria, minimizando así la posibilidad de aprovechar la superescalaridad del procesador y los beneficios de la consistencia relajada del acceso a memoria.

En este TFG se plantea la paralelización del algoritmo Matrix Profile usando el paradigma de Memoria Transaccional (particularmente, HTM) de manera que se pueda realizar una privatización parcial del profile a calcular, con un volcado final al profile global protegido por

transacción. De esta manera se busca minimizar el uso de recursos de memoria y el uso continuado de transacciones que puedan empeorar el rendimiento. El objetivo es conseguir reducir el consumo de memoria del algoritmo sin empeorar el tiempo de ejecución del mismo, a la vez que se facilita su paralelización. Además, se busca evaluar si es posible la optimización de los algoritmos ya existentes, estudiando si es factible el uso de transacciones en el cálculo del Matrix Profile.

1.2. Objetivos

Una serie temporal es una secuencia de muestras de una variable, generalmente de tipo real, recogidas y ordenadas en el tiempo. El algoritmo Matrix Profile [21] tiene como entrada una serie temporal, que puede ser de millones de muestras, y un tamaño de ventana, generalmente muy pequeño (unas cuantas muestras) respecto del tamaño de la serie. Con estos datos de entrada, el algoritmo divide la serie en ventanas y las compara todas dos a dos por medio del cálculo de una distancia. El resultado es un vector llamado Matrix Profile (del tamaño de la serie menos el tamaño de la ventana) que almacena en cada elemento la mínima distancia resultante de comparar la ventana correspondiente a ese elemento con todas las demás, y almacena en otro vector (del mismo tamaño) el índice dentro de la serie de la ventana con la que hace el mínimo. El algoritmo debe realizar un elevadísimo número de cálculos cuando las series son largas y requiere de paralelización para obtener resultados en un tiempo razonable. En la bibliografía podemos encontrar implementaciones para memoria distribuida y para GPU [22, 24] que requieren de la replicación de las estructuras de datos y de una programación muy compleja para el caso del procesador gráfico. También podemos encontrar una implementación para memoria compartida en una arquitectura manycore [12] haciendo uso de privatización. En este TFG se propone el uso de Memoria Transaccional (TM) [13, 14] para la paralelización de distintas versiones del algoritmo Matrix Profile. El paradigma de TM busca simplificar la programación paralela y mejorar el rendimiento introduciendo el concepto de transacción. Una transacción es un trozo de código que se ejecuta de manera atómica y aislada de otras transacciones a la vez que en paralelo a ellas (se dice que es un paradigma de paralelismo optimista). El sistema transaccional (software o hardware) se encarga de detectar dependencias entre transacciones y serializar la ejecución en caso de conflicto. Usando transacciones hardware (HTM), se plantea minimizar el uso de recursos de memoria del algoritmo

Matrix Profile y mantener su rendimiento utilizando el número de transacciones adecuado a la vez que se encuentra un tamaño óptimo de las mismas. Los objetivos de este TFG son los siguientes:

- Reducir el uso de recursos y la complejidad de programación del algoritmo Matrix Profile para series de gran tamaño con el uso del paradigma de programación paralela de Memoria Transaccional, a la vez que se mantiene el rendimiento.
- Ofrecer una solución de privatización parcial con volcado de datos parciales al array global protegido por transacción, que sea aplicable a otros códigos. Se puede estudiar la incorporación de la solución a una librería basada en pragmas como OpenMP.
- Aplicar el paradigma de TM a un problema real y hacer uso de un sistema HTM como el que dispone Intel.
- Hacer un estudio de la sensibilidad del tamaño de transacción de la solución de privatización parcial.
- Evaluar si es posible la optimización de los algoritmos ya existentes, estudiando si es factible el uso de transacciones en el cálculo del Matrix Profile.

1.3. Estado del arte

La memoria transaccional hardware (HTM) está disponible en productos de los principales proveedores. IBM fue el primero en ofrecer HTM en BlueGene/Q y el servidor zEnterprise EC12. En 2014, Intel anunció la inclusión de las extensiones de sincronización transaccional (TSX) en su arquitectura de microprocesador Haswell. Los beneficios de la simulación de software de esta tecnología seguirán siendo significativos incluso después de que los procesadores que permitan estas nuevas instrucciones estén disponibles en el mercado. La razón de esto es que una simulación a menudo proporciona más flexibilidad durante la depuración y la exploración de la arquitectura.

Por otro lado, el descubrimiento de patrones en series temporales es muy requerido para el análisis de series temporales y se utiliza en dominios como la neurociencia, la música y el análisis deportivo. En los últimos años, los avances algorítmicos (junto con las mejoras

en hardware) han ampliado en gran medida el alcance del descubrimiento de patrones. No obstante, sigue existiendo una necesidad de mayor escalabilidad.

El descubrimiento de patrones (repetidos) en series temporales es uno de los estudios en la minería de datos de series temporales. Estos patrones también se utilizan para la clasificación, agrupamiento, segmentación, visualización y detección de anomalías en algoritmos. Matrix Profile permite el cálculo exacto y eficiente de los top-k patrones de una serie temporal. Los algoritmos de última generación para calcular el Matrix Profile son lo suficientemente rápidos para muchas tareas. Sin embargo, en muchos dominios, incluyendo la astronomía y sismología, sigue existiendo la necesidad de considerar conjuntos de datos mucho más grandes.

1.4. Tecnologías y arquitecturas utilizadas

1.4.1. C

C es un lenguaje de programación de propósito general y de alto nivel que es ideal para desarrollar firmware o aplicaciones portables y fue diseñado originalmente para escribir software de sistema. C fue desarrollado en Bell Labs por Dennis Ritchie para el sistema operativo Unix a principios de la década de 1970.

Clasificado entre los lenguajes más utilizados, C tiene un compilador para la mayoría de los sistemas informáticos y ha influido en muchos lenguajes populares, en particular C++.

C pertenece a los paradigmas estructurados y procedurales de los lenguajes. Es flexible y puede usarse para una variedad de aplicaciones diferentes. Aunque sea un lenguaje de alto nivel, C y el lenguaje ensamblador comparten muchos de los mismos atributos.

1.4.2. C++

C++ es un lenguaje de programación orientado a objetos creado por Bjorne Stroustrop como parte de la evolución de la familia de lenguajes C. Fue desarrollado como una mejora multiplataforma de C para proporcionar a los desarrolladores un mayor grado de control sobre la memoria y los recursos del sistema.

Introduce principios de programación orientada a objetos, incluido el uso de clases definidas, en el marco del lenguaje de programación C.

Hoy en día, C++ sigue siendo muy apreciado por su notable portabilidad que permite a

los desarrolladores crear programas que pueden ejecutarse en diferentes sistemas operativos o plataformas con mucha facilidad. A pesar de ser un lenguaje de alto nivel, dado que C++ es a similar C, puede usarse para la manipulación de bajo nivel debido a su estrecha relación con el lenguaje de máquina.

1.4.3. Python

Python es un lenguaje de programación multiparadigma, de propósito general, interpretado y de alto nivel. Python permite a los programadores usar diferentes estilos de programación para crear programas simples o complejos, obtener resultados más rápidos y escribir código casi como si hablaran en un lenguaje humano.

El desarrollo inicial de Python fue encabezado por Guido van Rossum a fines de la década de 1980. Hoy en día, es desarrollado por Python Software Foundation. Debido a que Python es un lenguaje multiparadigma, los programadores de Python pueden realizar sus tareas utilizando diferentes estilos de programación: orientada a objetos, imperativa, funcional o reflexiva. Python se puede usar en desarrollo web, programación numérica, desarrollo de juegos, acceso a puerto serie y más.

1.4.4. LaTex

LaTeX es un sistema de preparación de documentos para composición tipográfica. Es el estándar de facto para la comunicación y publicación de documentos en la comunidad científica y es ampliamente utilizado por matemáticos, científicos, ingenieros, lingüistas e investigadores. El sistema facilita la integración de fórmulas matemáticas complejas, ecuaciones, bibliografías e índices en un documento.

LaTex es un paquete que permite a los usuarios utilizar el programa de composición tipográfica TeX como motor de formato. La composición tipográfica TeX está diseñada para documentar notaciones matemáticas complejas, texto y fórmulas. Es por eso que LaTeX es más adecuado para revistas técnicas, informes, libros y presentaciones de diapositivas. Una salida de LaTeX suele estar en un formato de archivo independiente del dispositivo, que luego se puede exportar a postscript o PDF.

Se ha utilizado LaTex para el desarrollo de esta memoria.

1.4.5. Arquitecturas

Para realizar los experimentos se ha utilizado un servidor equipado con 2 procesadores Intel Xeon Gold 6154 [3].

Este servidor dispone de las nuevas instrucciones introducidas en Intel Transactional Synchronization Extensions (Intel TSX), necesarias para utilizar Restricted Transactional Memory (RTM).

1.4.6. RTM

RTM (Restricted Transactional Memory) es una implementación alternativa a HLE (Hardware Lock Elision) que brinda al programador la flexibilidad de especificar una ruta de código de respaldo que se ejecuta cuando una transacción no se puede ejecutar con éxito. A diferencia de HLE, RTM no es compatible con procesadores que no lo admitan. Para la compatibilidad con versiones anteriores, los programas deben detectar la compatibilidad con RTM en la CPU antes de usar las nuevas instrucciones.

RTM agrega tres nuevas instrucciones: XBEGIN, XEND y XABORT. Las instrucciones XBEGIN y XEND marcan el inicio y el final de una región de código transaccional; XABORT aborta explícitamente una transacción. El aborto de la transacción redirige al procesador a la ruta del código de respaldo especificada en la instrucción XBEGIN con el estado de aborto devuelto en el registro EAX.

1.5. Metodología

Al tratarse de un trabajo esencialmente experimental nos basaremos en el método científico para desarrollar el TFG, centrándonos más específicamente en los procesos de experimentación, medición y análisis de resultados para extraer las hipótesis y conclusiones. Cabe resaltar también la característica de reproducibilidad del método científico, por lo que se tendrá especial cuidado en el diseño y programación de los experimentos y en su debida documentación.

En cuanto al desarrollo del software objeto del TFG, se trata de codificar un algoritmo ya existente usando tecnologías modernas de paralelización, y de implantar implementaciones ya existentes de este algoritmo para la posterior experimentación. Por lo tanto, no se ve la necesidad de recurrir a una metodología de desarrollo de software integral, como la de Proce-

so Unificado (basada en UML), por ejemplo, con todas sus fases. Sin embargo, se debe hacer uso de metodologías de la programación (especificación del problema, diseño, codificación, documentación, ...) y pruebas del software para el adecuado desarrollo de la programación.

1.6. Estructura de la memoria

Se han descrito en este primer capítulo (introducción), los motivos y objetivos de por qué se ha decidido realizar este TFG. Seguido del estado del arte, las tecnologías, arquitectura utilizada y la metodología.

En el Capítulo 2, conocimientos previos, se describe la información necesaria para entender como se ha implementado los algoritmos que se van a utilizar para experimentar.

En la Capítulo 3, implementaciones, se muestran los distintos algoritmos y diferentes versiones de forma detallada.

En la Capítulo 4, fase de experimentos y resultados, se recoge los resultados de las pruebas realizadas y se hace una breve conclusión.

En la Capítulo 5 y final, conclusiones, donde se finaliza la memoria con las conclusiones finales de este TFG.

2

Conocimientos previos

El cálculo del Matrix Profile resuelve el problema de all-pairs-similarity-search (también conocido como similarity join)[9, 10, 11, 18, 20, 21, 22, 23, 24]. Este problema se reduce en: dada una colección de objetos de datos, recuperar el vecino más cercano para cada objeto. La estructura que almacena la información del vecino más cercano para cada objeto se llama Matrix Profile.

En este TFG se ha decidido experimentar con los algoritmos SCRIMP [22] y SCAMP [24]. Ambos algoritmos calculan el Matrix Profile y están inspirados en algoritmos anteriores como STAMP o STOMP [21]. A la hora de calcular el Matrix Profile se divide la serie en ventanas y las compara todas dos a dos por medio del cálculo de una distancia. Los algoritmos de diferencian en la forma de calcular esta distancia y en el cálculo de unas variables estadísticas necesarias para obtener dicha distancia.

2.1. Matrix Profile

Primero introducimos una serie de definiciones formales que SCRIMP y SCAMP utilizan [21, 22, 24]:

Definición 1: Una serie temporal T es una secuencia de números reales $t_i : T = t_1, t_2, \ldots, t_n$ donde n es la longitud de T.

Definición 2: Una subsecuencia $T_{i,m}$ de una serie temporal T es un subconjunto continuo de valores de T de tamaño m empezando en la posición i. Es decir, $T_{i,m} = t_i, t_{i+1}, \ldots, t_{i+m-1}$ donde $1 \leq i \leq n - m + 1$.

Definición 3: Un $Distance\ Profile\ D_i$ correspondiente a una subsecuencia $T_{i,m}$ y una serie temporal T, es un vector de las distancias euclídeas entre la subsecuencia dada $T_{i,m}$ y cada subsecuencia en la serie temporal T. Formalmente, $D_i = [d_{i,1}, d_{i,2}, \ldots, d_{i.n-m+1}]$, donde $d_{i,j}(1 \leq j \leq n - m + 1)$ es la distancia entre $T_{i,m}$ y $T_{j,m}$.

Una vez obtenido D_i, podemos calcular la subsecuencia más próxima de $T_{i,m}$ en T. Destacar que si $T_{i,m}$ es una subsecuencia de T, la posición i del Distance Profile D_i es cero ($d_{i,i} = 0$) y se le llama $trivial\ match$. Éstos son ignorados utilizando una zona de "exclusión" de tamaño $m/4$ antes y después de la posición i. En la práctica $d_{i,j}(i - m/4 \leq j \leq i + m/4)$ son inicializados a infinito y la subsecuencia más cercana de $T_{i,m}$ se obtiene evaluando $min(D_i)$.

Se necesita encontrar la subsecuencia más cercana de cada subsecuencia de T. La información de la subsecuencia más cercana es almacenada en dos vectores, el $Matrix\ Profile$ y el $Matrix\ Profile\ index$.

Definición 4: Un $Matrix\ Profile\ P$ de una serie temporal T es un vector de las distancias euclídeas entre cada subsecuencia de T y su subsecuencia más próxima en T. Formalmente, $P = [min(D_1), min(D_2), \ldots, min(D_{n-m+1})]$, donde D_i $(1 \leq i \leq n - m + 1)$ es el Distance Profile D_i correspondiente a $T_{i,m}$ y a la serie temporal T.

La posición i del Matrix Profile P almacena la distancia entre la subsecuencia $T_{i,m}$ y su subsecuencia más próxima dentro de la serie temporal T. Sin embargo, no almacena la posición de la subsecuencia más cercana; esta posición es almacenada en el $Matrix\ Profile\ index$.

Definición 5: Un $Matrix\ Profile\ index\ I$ de una serie temporal es un vector de enteros $I = [I_1, I_2, \ldots, I_{n-m+1}]$, donde $I_i = j$ si $d_{i,j} = min(D_i)$.

	D_1	D_2	...	D_{n-m+1}
D_1	$d_{1,1}$	$d_{1,2}$	...	$d_{1,n-m+1}$
D_2	$d_{2,1}$	$d_{2,2}$	...	$d_{2,n-m+1}$
...	...	...	...	...
D_{n-m+1}	$d_{n-m+1,1}$	$d_{n-m+1,2}$	...	$d_{n-m+1,n-m+1}$
	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$
P	$min(D_1)$	$min(D_2)$	...	$min(D_{n-m+1})$

Figura 1: Relación entre la Distance Matrix, Distance Profile y Matrix Profile.

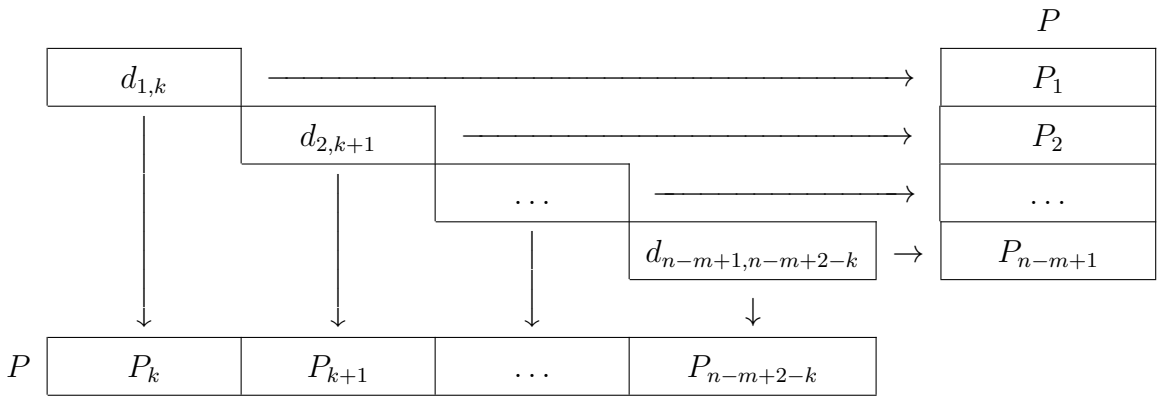

Figura 2: Actualización del Matrix Profile en cada iteración.

El Distance Profile es una columna (o fila) de la Distance Matrix (Fig. 1). El Matrix Profile almacena el mínimo valor, ignorando la diagonal, de cada columna de la Distance Matrix y la posición correspondiente a dicho valor se almacena en el Matrix Profile index.

En la práctica se calcula una distancia $d_{i,j}$,entre las subsecuencias $T_{i,m}$ y $T_{j,m}$, y se comprueba si es menor que la distancia almacenada actualmente en el Matrix Profile en dos posiciones P_i y P_j. Por lo que no es requerido crear una estructura para la Distance Matrix.

La Figura 2 muestra una posible diagonal donde se calcula la distancia y se almacena en el Matrix Profile si la distancia es menor.

2.1.1. SCRIMP

El algoritmo SCRIMP determina la distancia $d_{i,j}$ de dos subsecuencias $T_{i,m}$ y $T_{j,m}$ de la serie temporal T utilizando la Ecuación 1.

$$d_{i,j} = \sqrt{2m\left(1 - \frac{Q_{i,j-m} - m\mu_i\mu_j}{m\sigma_i\sigma_j}\right)} \tag{1}$$

Donde m es el tamaño de la subsecuencia, $Q_{i,j}$ es el producto escalar de $T_{i,m}$ y $T_{j,m}$, μ_i es la media de $T_{i,m}$, μ_j es la media de $T_{j,m}$, σ_i es la desviación típica de $T_{i,m}$ y σ_j es la desviación típica de $T_{j,m}$.

Podemos precalcular las medias y desviaciones típicas en un tiempo O(n) aplicando una técnica explicada en [21, 22, 24]. Una vez hecho el paso previo obtenemos las medias y desviaciones típicas en un tiempo O(1). También se demuestra en [21, 22, 24] que $Q_{i,j}$ se puede

calcular en un tiempo O(1) después de calcular $Q_{i-1,j-1}$, utilizando la Ecuación 2.

$$Q_{i,j} = Q_{i-1,j-1} - t_{i-1}tj - 1 + t_{i+m-1}t_{j+m-1} \tag{2}$$

Teniendo en cuenta las Ecuaciones 1 y 2, se podría calcular la Distance Matrix (Fig. 1) fila por fila y luego actualizar el Matrix Profile en un tiempo de O(n^2). Pero la Ecuación 2 nos permite evaluar las diagonales de la Distance Matrix consiguiendo encontrar posibles patrones al final de la serie temporal en una etapa inicial, que en otro caso habría que esperar a la finalización del algoritmo. Podemos ver una iteración del algoritmo en la Figura 2.

2.1.2. SCAMP

En cambio, el algoritmo SCAMP ha modificado los cálculos de punto flotante y el producto escalar utilizados en la Ecuación 1 por las Ecuaciones 3, 4, 5 y 6, siendo éstas, fórmulas de sumas de productos.

$$df_0 = 0; df_i = \frac{T_{i+m-1} - T_{i-1}}{2} \tag{3}$$

$$dg_0 = 0; dg_i = (T_{i+m-1} - \mu_i) + (T_{i-1} - \mu_{i-1}) \tag{4}$$

$$\overline{QT}_{i,j} = \overline{QT}_{i-1,j-1} + df_i dg_j + df_j dg_i \tag{5}$$

$$P_{i,j} = \overline{QT}_{i,j} * \frac{1}{\|T_{i,m} - \mu_i\|} * \frac{1}{\|T_{j,m} - \mu_j\|} \tag{6}$$

$$d_{i,j} = \sqrt{2m\,(1 - P_{i,j})} \tag{7}$$

Las Ecuaciones 3, 4 y la media son valores precalculados que serán, posteriormente, utilizados en la Ecuación 5. SCAMP remplaza la distancia euclídea por el coeficiente de correlación de Pearson, aunque puede ser convertido a ésta utilizando la Ecuación 7.

2.2. RTM

Restricted Transactional Memory (RTM) es una de las interfaces que proporciona Intel Transactional Synchronization Extensions (Intel TSX) [15, 16] con la intención de mejorar el rendimiento de las regiones críticas protegidas por locks mientras se mantiene el mismo modelo de programación.

Intel TSX permite al procesador determinar dinámicamente que threads necesitan serializar la ejecución de las regiones críticas protegidas por locks y únicamente hacerlo cuando sea necesario. Esto permite explorar y explotar concurrencia "escondida" en la aplicación debido a sincronización innecesaria. Estas regiones críticas se les llama transacción. Una transacción es un trozo de código que se ejecuta de manera atómica y aislada de otras transacciones a la vez que en paralelo a ellas.

Si la transacción finaliza con éxito, todas las operaciones hechas en la memoria, ejecutadas dentro de la transacción, serán visibles instantáneamente para el resto de procesadores. A este proceso se le llama atomic commit.

El procesador ejecuta el código de forma optimista sin sincronización, ya que si el procesador no puede realizar atomic commit, la ejecución optimista falla y el procesador vuelve atrás en la ejecución. A este proceso se le llama aborto de la transacción (o transactional abort). Cuando esto ocurre el procesador descarta todas las operaciones realizadas dentro de la transacción, restaura el estado inicial como si el proceso optimista nunca hubiese ocurrido y continúa la ejecución de forma no transaccional.

Cuando se utiliza la interfaz RTM, se debe proporcionar un camino no transaccional (en el que eventualmente se adquiera el lock que estamos evitando usar en un primer momento) para ejecutar en caso de que la transacción aborte. La aplicación no debe depender únicamente de la ejecución transaccional.

Las instrucciones introducidas son las siguientes:

- **_xtest :** Pregunta si el procesador está ejecutando una región transaccional.
- **_xbegin :** Especifica el comienzo de la transacción y devuelve un valor indicando si la transacción se ha iniciado correctamente o el estado de aborto de la transacción. Este estado se almacena en un registro EAX. Declara un offset para hacer fallback después de un aborto, continuando la ejecución en la siguiente instrucción.

- **_xend :** Especifica el final de la transacción y el procesador intenta hacer commit de las operaciones realizadas dentro de la transacción. En caso de fallar, el procesador hace rollback, descarta todas las operaciones y hace fallback continuando en la siguiente instrucción después de _xbegin.

- **_xabort :** Fuerza la transacción a abortar con el estado proporcionado por argumento y se actualiza el registro EAX con el estado indicado.

Utilizando el estado almacenado en el registro EAX podemos personalizar la ejecución en función del motivo del aborto. Por ejemplo si la transacción aborta porque otro thread adquirió el lock, podemos esperar a que se libere y reintentar ejecutar la transacción.

Intel proporciona una serie de motivos por los que una transacción puede abortar [15].

2.2.1. API C / C++

Las implementaciones de los algoritmos SCRIMP y SCAMP se han realizado utilizando C y C++ [8, 17]. C dispone de una librería para utilizar los conceptos de RTM:

- **immintrin.h :** Librería en la que se definen las instrucciones necesarias para utilizar RTM. Se definen los siguientes macros para diferenciar motivos de abortos:

  - **_XABORT_EXPLICIT :** La transacción ha abortado explícitamente utilizando _xabort.
  - **_XABORT_RETRY :** La transacción puede ser exitosa si se reintenta.
  - **_XABORT_CONFLICT :** La transacción ha abortado por un conflicto en memoria con otro thread.
  - **_XABORT_CAPACITY :** La transacción ha abortado porque utiliza mucha memoria (o porque no existen los suficientes recursos).
  - **_XABORT_DEBUG :** La transacción ha abortado por una excepción de debug.
  - **_XABORT_NESTED :** La transacción ha abortado en un anillo interior.

3

Implementaciones

En esta sección, se muestran las diferentes versiones implementadas comenzando con las versiones que utilizan RTM, introduciendo las transacciones explicadas en las Secciones 2.2 y 2.2.1, seguido de las versiones que utilizan el control de acceso a variables compartidas tradicional a través de locks y por último la versión más utilizada a día de hoy que no requiere de un control de acceso de la variable global, ya que privatiza las estructuras globales replicándolas por cada thread. Destacar que las versiones donde utilizamos RTM y locks convencionales, al no ser necesario la duplicación de las estructuras globales Matrix Profile y Matrix Profile index por cada thread utilizado, se consigue reducir el uso de memoria de forma considerable.

3.1. RTM-UniTransaccion

RTM-UniTransaccion es la primera versión del algoritmo y se caracteriza por utilizar el concepto de transacciones RTM y realizar una única actualización (lectura y posible escritura) de las estructuras Matrix Profile y Matrix Profile index por cada transacción.

Se ha organizado el programa en 3 archivos diferentes:

- **SCRIMP.cpp/SCAMP.cpp :** Archivo principal del algoritmo que tiene como entrada el nombre del fichero de la serie temporal, el tamaño de la ventana a utilizar y el número de threads, y como salida un excel con las estructuras Matrix Profile y Matrix Profile index.

- **transaction.h :** Archivo de declaración de funciones utilizadas para controlar el manejo de las transacciones RTM y obtener estadísticas sobre éstas.

- **transaction.c :** Archivo de implementación de funciones de transaction.h donde se controla el manejo de las transacciones RTM y se obtienen estadísticas sobre éstas.

3.1.1. SCRIMP.cpp

El archivo declara las siguientes variables globales:

- **PATH_RESULTS :** Directorio a guardar archivos de salida.
- **SHUFFLE_DIAGS :** Opción para mezclar diagonales para habilitar anytime algorithm.
- **numThreads** $\langle long\ long\ int \rangle$ **:** Dato de entrada que determina el número de threads utilizados.
- **exclusionZone** $\langle long\ long\ int \rangle$ **:** Valor utilizado para ignorar los trivial match. La exclusion zone recomendada es 1/4 del tamaño de ventana.
- **windowSize** $\langle long\ long\ int \rangle$ **:** Dato de entrada que determina el tamaño de las subsecuencias.
- **tSeriesLength** $\langle long\ long\ int \rangle$ **:** Tamaño de la serie temporal introducida como dato de entrada.
- **profileLength** $\langle long\ long\ int \rangle$ **:** Tamaño del Matrix Profile calculada a partir de tSeriesLength y windowSize como: $tSeriesLength - windowSize + 1$.

El archivo contiene las siguientes funciones:

main() Función donde comienza la ejecución del programa (Código 1). Empezando con la lectura de los datos de entrada hasta la escritura de las estructuras Matrix Profile y Matrix Profile index en un archivo excel de salida. Las variables declaradas en esta función son :

- **tstart** $\langle chrono :: steady\_clock :: time\_point \rangle$ **:** Variable utilizada para calcular tiempos de ejecución.
- **tprogstart** $\langle chrono :: steady\_clock :: time\_point \rangle$ **:** Variable utilizada para calcular tiempo de ejecución total del programa.
- **tend** $\langle chrono :: steady\_clock :: time\_point \rangle$ **:** Variable utilizada para calcular tiempos de ejecución.

- **telapsed** $\langle chrono :: duration \langle double \rangle \rangle$ **:** Variable utilizada para calcular tiempos de ejecución.
- **inputfilename** $\langle string \rangle$ **:** Dato de entrada que determina la ruta del fichero que contiene la serie temporal.
- **outfilename** $\langle string \rangle$ **:** Nombre de fichero de salida a guardar el Matrix Profile y Matrix Profile index.
- **tSeriesFile** $\langle fstream \rangle$ **:** Stream utilizado para lectura de los datos del fichero almacenado en $inputfilename$
- **tSeries** $\langle vector \langle double \rangle \rangle$ **:** Vector que almacena la serie temporal.
- **means** $\langle vector \langle double \rangle \rangle$ **:** Vector que almacena las medias calculadas.
- **devs** $\langle vector \langle double \rangle \rangle$ **:** Vector que almacena las desviaciones típicas calculadas.
- **profile** $\langle vector \langle double \rangle [profileLength] \rangle$ **:** Vector que almacena la estructura Matrix Profile
- **diags** $\langle vector \langle long\ long\ int \rangle \rangle$ **:** Vector que almacena las diagonales a recorrer.
- **profileIndex** $\langle vector \langle long\ long\ int \rangle [profileLength] \rangle$ **:** Vector que almacena la estructura Matrix Profile index.
- **statsFile** $\langle fstream \rangle$ **:** Stream utilizado para escritura de las estructuras Matrix Profile y Matrix Profile index en el fichero almacenado en $outfilename$
- **memoria** $\langle int \rangle$ **:** Variable utilizada para el cálculo de la memoria utilizada en esta versión para su próxima comparación con el resto de versiones.
- **tiempo** $\langle char^* \rangle$ **:** Variable utilizada para almacenar el tiempo de ejecución de la función SCRIMP().

```
''_
int main(int argc, char *argv[])
{
  try
  {

    if (argc != 4)
    {
      cout << "[ERROR] usage: ./scrimp input_file win_size num_threads" << endl;
      return 0;
    }

    // Creacion de variables utilizadas para la medida del tiempo
    chrono::steady_clock::time_point tstart, tprogstart, tend;
    chrono::duration<double> telapsed;

    // Actualizacion de los parametros del algoritmo utilizando los datos de entrada
    windowSize = atoi(argv[2]);
    numThreads = atoi(argv[3]);
    exclusionZone = (ITYPE)(windowSize * 0.25);     // La exclusion zone recomendada es
    1/4 del tamaño de ventana

    omp_set_num_threads(numThreads);


    string inputfilename = argv[1];
    stringstream tmp;
    tmp << PATH_RESULTS << argv[0] << "_" << inputfilename.substr(inputfilename.rfind('/'
    ) + 1, inputfilename.size() - 4 - inputfilename.rfind('/') - 1) <<
                  "_w" << windowSize << "_t" << numThreads << "_" << getpid() << ".csv";
    string outfilename = tmp.str(); // Nombre de fichero de salida


    cout << endl;
    cout << "####################################################################" << endl;
    cout << "////////////////////////// SCRIMP ///////////////////////////////" << endl;
    cout << "####################################################################" << endl;
    cout << endl;

    // Lectura de datos de la serie temporal
    cout << "[>>] Reading File... " << inputfilename << endl;

    tstart = chrono::steady_clock::now();
    tprogstart = tstart;

    fstream tSeriesFile(inputfilename, ios_base::in);

    vector<DTYPE> tSeries;
    DTYPE tempval, tSeriesMin = numeric_limits<DTYPE>::infinity(), tSeriesMax = -
    numeric_limits<double>::infinity();

    tSeriesLength = 0;
    while (tSeriesFile >> tempval)
    {
      tSeries.push_back(tempval);
      if (tempval < tSeriesMin)
        tSeriesMin = tempval;
      if (tempval > tSeriesMax)
        tSeriesMax = tempval;
      tSeriesLength++;
    }
    tSeriesFile.close();
```

```
    tend = chrono::steady_clock::now();
    telapsed = tend - tstart;

    cout << "[OK] Read File Time: " << setprecision(numeric_limits<double>::digits10 + 2)
     << telapsed.count() << " seconds." << endl;

    // Calculo del tamaño de la matrix profile
    profileLength = tSeriesLength - windowSize + 1;

    vector<DTYPE> means, devs, profile(profileLength);
    vector<ITYPE> diags, profileIndex(profileLength);


    cout << endl;
    cout << "------------------------------------------------------------" << endl;
    cout << "************************* INFO ****************************" << endl;
    cout << endl;
    cout << " Series/MP data type: " << typeid(tSeries[0]).name() << "(" << sizeof(
    tSeries[0]) << "B)" << endl;
    cout << " Index data type:     " << typeid(profileIndex[0]).name() << "(" << sizeof(
    profileIndex[0]) << "B)" << endl;
    cout << " Time series length:  " << tSeriesLength << endl;
    cout << " Window size:         " << windowSize << endl;
    cout << " Exclusion zone:      " << exclusionZone << endl;
    cout << " Profile length:      " << profileLength << endl;
    cout << " Number of threads:   " << numThreads << endl;
    cout << " Time series min:     " << tSeriesMin << endl;
    cout << " Time series max:     " << tSeriesMax << endl;
    cout << " Random diag. order:  ";
    if (SHUFFLE_DIAGS)
      cout << "true" << endl;
    else
      cout << "false" << endl;
    cout << endl;
    cout << "------------------------------------------------------------" << endl;
    cout << endl;

    // Inicializacion de matrix profile
    cout << "[>>] Initializing Profile..." << endl;

    tstart = chrono::steady_clock::now();

    for (uint64_t i = 0; i < (uint64_t) profileLength; i++)
      profile[i] = numeric_limits<DTYPE>::infinity();

    tend = chrono::steady_clock::now();
    telapsed = tend - tstart;

    cout << "[OK] Initializing Profile Time: " << setprecision(numeric_limits<DTYPE>::
    digits10 + 2) << telapsed.count() << " seconds." << endl;

    // Preproceso para calcular la media y la desviacion tipica de cada subsecuencia
    utilizaos para el calculo de la distancia
    cout << "[>>] Preprocessing..." << endl;

    tstart = chrono::steady_clock::now();

    preprocess(tSeries, means, devs);

    tend = chrono::steady_clock::now();
    telapsed = tend - tstart;

    cout << "[OK] Preprocessing Time:         " << setprecision(numeric_limits<double>::
```

```
    digits10 + 2) << telapsed.count() << " seconds." << endl;

    // Calculo de diagonales y mezcla aleatoria en caso de utilizar anytime algorithm
    diags.clear();
    for (ITYPE i = exclusionZone + 1; i < profileLength; i++)
      diags.push_back(i);

    if (SHUFFLE_DIAGS)
      random_shuffle(diags.begin(), diags.end());

    // StatsFileInit para inicializar estrucutura de estadisticas y ticketlock
    stringstream tempf;
    tempf<<outfilename.substr(0,outfilename.size()-4)<<".stats";
    string f=tempf.str();
    char *cstr = new char[f.length() + 1];
    strcpy(cstr, f.c_str());

    statsFileInit(cstr,numThreads,1,profileLength);

    // Scrimp
    cout << "[>>] Executing SCRIMP..." << endl;

    tstart = chrono::steady_clock::now();

    scrimp(tSeries, diags, means, devs, profile, profileIndex);

    tend = chrono::steady_clock::now();
    telapsed = tend - tstart;

    cout << "[OK] SCRIMP Time:             " << setprecision(numeric_limits<DTYPE>::
    digits10 + 2) << telapsed.count() << " seconds." << endl;

    stringstream tiempo_temp;
    tiempo_temp<<setprecision(numeric_limits<double>::digits10 + 2) <<telapsed.count();
    string tiempo_temp2=tiempo_temp.str();
    char *tiempo= new char[tiempo_temp2.length() +1];
    strcpy(tiempo,tiempo_temp2.c_str());

    // Guardando profile
    cout << "[>>] Saving Profile..." << endl;

    tstart = chrono::steady_clock::now();

    fstream statsFile(outfilename, ios_base::out);

    statsFile << "# Time (s)" << endl;
    statsFile << setprecision(9) << telapsed.count() << endl;

    statsFile << "# Profile Length" << endl;
    statsFile << profileLength << endl;

    statsFile << "# i;tseries;profile;index" << endl;
    for (ITYPE i = 0; i < profileLength; i++)
    {
      statsFile << i << ";" << tSeries[i] << ";" << (DTYPE)sqrt(profile[i]) << ";" <<
    profileIndex[i] << endl;
    }
    statsFile.close();

    tend = chrono::steady_clock::now();
    telapsed = tend - tstart;

    cout << "[OK] Saving Profile Time:       " << setprecision(numeric_limits<DTYPE>::
```

```
    digits10 + 2) << telapsed.count() << " seconds." << endl;

    // Calcular tiempo total
    telapsed = tend - tprogstart;

    cout << "[OK] Total Time:              " << setprecision(numeric_limits<DTYPE>::
    digits10 + 2) << telapsed.count() << " seconds." << endl;
    cout << endl;

    //DTYPE(double):   8B; ITYPE(long long int): 8B; profile y profileIndex de tamaño
    profileLength ambas
    int memoria=(sizeof(DTYPE)+sizeof(ITYPE))*profileLength;

    // Volcado de estadisticas en fichero de salida
    dumpStats(tiempo,memoria);
    delete tiempo;
    delete cstr;
  }
  catch (exception &e)
  {
    cout << "Exception: " << e.what() << endl;
  }
}
```

Código 1: Función main() del archivo SCRIMP.cpp de la versión RTM-UniTransaccion.

preprocess() : Función para el cálculo de las medias y desviaciones típicas (Código 2), utilizadas posteriormente en la función SCRIMP(). Las variables declaradas en esta función son:

- **ACumSum** $\langle *double[tSeriesLength] \rangle$ **:** Variable que almacena las sumas acumulativas de la serie temporal.
- **ASqCumSum** $\langle *double[tSeriesLength] \rangle$ **:** Variable que almacena las sumas cuadráticas acumulativas de la serie temporal.
- **ASum** $\langle *double[profileLength] \rangle$ **:** Variable que almacena las sumas de las subsecuencias.
- **ASumSq** $\langle *double[profileLength] \rangle$ **:** Variable que almacena las sumas cuadráticas de las subsecuencias.
- **ASigmaSq** $\langle *double[profileLength] \rangle$ **:** Variable que almacena las varianzas de las subsecuencias.

```
''_
// Calculo de valores estadisticos utilizados en scrimp
void preprocess(vector<DTYPE> &tSeries, vector<DTYPE> &means, vector<DTYPE> &devs)
{
  DTYPE *ACumSum = new DTYPE[tSeriesLength];
  DTYPE *ASqCumSum = new DTYPE[tSeriesLength];
  DTYPE *ASum = new DTYPE[profileLength];
  DTYPE *ASumSq = new DTYPE[profileLength];
  DTYPE *ASigmaSq = new DTYPE[profileLength];

  means.clear();
  devs.clear();

  // Sumas acumulativas de la serie temporal
  ACumSum[0] = tSeries[0];
  ASqCumSum[0] = tSeries[0] * tSeries[0];
  for (ITYPE i = 1; i < tSeriesLength; i++)
  {
    ACumSum[i] = tSeries[i] + ACumSum[i - 1];
    ASqCumSum[i] = tSeries[i] * tSeries[i] + ASqCumSum[i - 1];
  }

  // Sumas de las subsecuencias
  ASum[0] = ACumSum[windowSize - 1];
  ASumSq[0] = ASqCumSum[windowSize - 1];
  for (ITYPE i = 0; i < tSeriesLength - windowSize; i++)
  {
    ASum[i + 1] = ACumSum[windowSize + i] - ACumSum[i];
    ASumSq[i + 1] = ASqCumSum[windowSize + i] - ASqCumSum[i];
  }

  // Calculo de media y desviacion tipica de las subsecuencias
  for (ITYPE i = 0; i < profileLength; i++)
  {
    means.push_back(ASum[i] / windowSize);
    ASigmaSq[i] = ASumSq[i] / windowSize - means[i] * means[i];
    devs.push_back(sqrt(ASigmaSq[i]));
  }

  delete ACumSum;
  delete ASqCumSum;
  delete ASum;
  delete ASumSq;
  delete ASigmaSq;
}
```

Código 2: Función preprocess() del archivo SCRIMP.cpp de la versión RTM-UniTrasanccion.

SCRIMP() : Función que calcula las estructuras Matrix Profile y Matrix Profile Index globales (Código 3). En esta función aplicamos los conocimientos de RTM y ejecutamos las transacciones, a través de las funciones INIT_TRANSACTION(), BEGIN_TRANSACTION() y COMMIT_TRANSACTION(), que se utiliza para actualizar las estructuras globales Matrix Profile y Matrix Profile index. Las variables declaradas en esta función son :

- **diag** $\langle long\ long\ int \rangle$ **:** Variable que almacena la columna inicial de la diagonal a recorrer.
- **dotProd** $\langle double \rangle$ **:** Variable que almacena el producto escalar de las subsecuencias.
- **j** $\langle long\ long\ int \rangle$ **:** Variable que almacena la columna actual de la diagonal a recorrer.
- **i** $\langle long\ long\ int \rangle$ **:** Variable que almacena la fila actual de la diagonal a recorrer.
- **distance** $\langle double \rangle$ **:** Variable que almacena la distancia entre las subsecuencias.
- **myid** $\langle int \rangle$ **:** Variable que almacena el id del thread actual utilizado para obtener estadísticas de cada thread.

```
''_
void scrimp(vector<DTYPE> &tSeries, vector<ITYPE> &idx, vector<DTYPE> &means, vector<
    DTYPE> &devs, vector<DTYPE> &profile, vector<ITYPE> &profileIndex)
{



#pragma omp parallel shared(profile,profileIndex)
  {
    INIT_TRANSACTION();
#pragma omp for schedule(dynamic)
    for (ITYPE ri = 0; ri < (ITYPE)idx.size(); ri++)
    {
      // Seleccion diagonal
      ITYPE diag = idx[ri];
      DTYPE dotProd = 0;

      // Calculo del primer dotProd de la diagonal
      for (ITYPE j = diag; j < windowSize + diag; j++)
        dotProd += tSeries[j] * tSeries[j - diag];

      // la i es la fila y j la columa de la matrix profile, cuya distancia se va a
    calcular
      ITYPE j = diag;
      ITYPE i = 0;

      // Calculo de la distancia
      DTYPE distance = 2 * (windowSize - (dotProd - windowSize * means[j] * means[i]) / (
    devs[j] * devs[i]));

      // Actualizacion del profile y profileIndex si la distancia actual es menor
      int myid=omp_get_thread_num();
```

```
      BEGIN_TRANSACTION(myid,0);
      if (distance < profile[j])
      {
        profile[j] = distance;
        profileIndex[j] = i;
      }
      COMMIT_TRANSACTION(myid,0);

      BEGIN_TRANSACTION(myid,0);
      if (distance < profile[i])
      {
        profile[i] = distance;
        profileIndex[i] = j;
      }
      COMMIT_TRANSACTION(myid,0);

      i = 1;

      // Calculo del resto de distancias en la diagonal
      for (ITYPE j = diag + 1; j < profileLength; j++)
      {
        dotProd += (tSeries[j + windowSize - 1] * tSeries[i + windowSize - 1]) - (tSeries
    [j - 1] * tSeries[i - 1]);
        distance = 2 * (windowSize - (dotProd - means[j] * means[i] * windowSize) / (devs
    [j] * devs[i]));

        BEGIN_TRANSACTION(myid,0);
        if (distance < profile[j])
        {
          profile[j] = distance;
          profileIndex[j] = i;
        }
        COMMIT_TRANSACTION(myid,0);
        BEGIN_TRANSACTION(myid,0);
        if (distance < profile[i])
        {
          profile[i] = distance;
          profileIndex[i] = j;
        }
        COMMIT_TRANSACTION(myid,0);

        i++;
      }
    } //'pragma omp for' asegura que a partir de aqui todos los threads han acabado


  }
}
```

Código 3: Función SCRIMP() del archivo SCRIMP.cpp de la versión RTM-UniTransaccion.

3.1.2. SCAMP.cpp

El programa SCAMP tiene una estructura muy similar a SCRIMP a excepción de que utilizamos el coeficiente de correlación de Pearson (Ecuación 6) en vez de la distancia euclídea (Ecuación 1). Por lo que la función preprocess() calcula unos valores estadísticos diferentes a los de SCRIMP. Las funciones modificadas son las siguientes:

main() : Similar a la función main() de SCRIMP.cpp a excepción de las siguientes variables:

- **norms** $\langle vector\ \langle double \rangle\ [profileLength] \rangle$ **:** Variable que almacena las L2-normas de las subsecuencias.
- **df** $\langle vector\ \langle double \rangle\ [profileLength] \rangle$ **:** Variable que almacena los df de las subsecuencias.
- **dg** $\langle vector\ \langle double \rangle\ [profileLength] \rangle$ **:** Variable que almacena los dg de las subsecuencias.

preprocess() : Función para el cálculo de df, dg, L2-norma y las medias, utilizadas posteriormente en la función SCAMP(). Las variables declaradas en esta función son :

- **prefix_sum** $\langle vector\ \langle double \rangle\ [tSeriesLength] \rangle$ **:** Variable que almacena las sumas acumulativas de la serie temporal.
- **prefix_sum_sq** $\langle vector\ \langle double \rangle\ [tSeriesLength] \rangle$ **:** Variable que almacena las sumas cuadrática acumulativa de la serie temporal.

SCAMP() : Similar a SCRIMP() con la diferencia que utiliza el coeficiente de correlación de Pearson (Ecuación 6) en lugar de la distancia euclídea (Ecuación 1).

3.1.3. transaction.h

En este archivo se declaran las funciones utilizadas para controlar el manejo de las transacciones RTM y obtener estadísticas sobre éstas [1, 2, 4, 7, 19]. Se definen las funciones necesarias para inicializar, iniciar y terminar una transacción, así como los reintentos permitidos en caso

de aborto y su posterior adquisición del ticketlock. Además se declaran las funciones utilizadas para contabilizar el número de abortos o commits realizados por cada thread. Así como las funciones que serán implementadas en el archivo transaction.c.

Además Intel detalla una serie de recomendaciones [15, 16] que han sido tomadas en cuenta para evitar que las transacciones aborten:

- Las funciones dentro de las transacciones han sido definidas como inline.
- Las estructuras de estadísticas han sido creadas para almacenar los datos de cada thread en vez de almacenar los datos en una estructura compartida. Posteriormente se suman para tener los datos globales del programa.
- Se ha añadido padding a las estructuras utilizadas por los threads para evitar false sharing, lo cual también ocasiona abortos de las transacciones. Las estructuras utilizadas son: Stats y TicketLock.
- Se permite a los threads 3 reintentos (podría ser otro número de reintentos) para ejecutar la región crítica de forma transaccional.
- Minimizar el código ejecutado únicamente de forma transaccional.
- Abortar la transacción utilizando xabort en el camino transaccional en caso de que otro thread haya adquirido el ticketlock.
- Intel también recomienda que en el código ejecutado de forma transaccional se compruebe el ticketlock.
- Se ha utilizado una función para actualizar el lock de forma atómica (__sync_add_and_fetch()).
- Hacer usos de spinlocks para esperar a que el thread que ha adquirido el ticketlock lo deje libre. Esto es necesario porque si un thread ejecuta la transacción mientras otro thread posee el ticketlock, la transacción abortará. Por lo que el thread gastaría todos los intentos para ejecutar la región de forma transaccional y acabaría esperando para adquirir el ticketlock, haciendo que la región nunca se pueda realizar de forma transaccional.

El archivo declara las siguientes variables globales:

- **LOCK_TAKEN :** Valor aportado a _xabort() para especificar que el aborto explícito de la transacción ha sido causada porque el lock está cogido.
- **CACHE_BLOCK_SIZE :** Tamaño utilizado para hacer padding en la estructura Stats y TicketLock para evitar false sharing.
- **GLOBAL_RETRIES :** Número de reintentos de realizar con éxito la transacción. Una vez gastado los reintentos se serializa y se utiliza el lock.
- **g_ticketlock** $\langle struct\ TicketLock \rangle$ **:** Lock utilizado para serializar la actualización de las estructuras globales al acabar los reintentos. Al utilizar el ticketLock podemos tener un orden entre los threads en espera.
- **stats** $\langle struct\ Stats\ ** \rangle$**:** Estructura utilizada para obtener estadísticas de las transacciones.

El archivo contiene las siguientes funciones:

INIT_TRANSACTION() : Función para inicializar las variables utilizadas en BEGIN_TRANSACTION() y COMMIT_TRANSACTION() (Código 4), siendo éstas las siguientes:

- **__p_status** $\langle unsigned\ long \rangle$ **:** Variable utilizada para almacenar el estado de la transacción obtenida por _xbegin().
- **__p_retries** $\langle unsigned\ long \rangle$ **:** Variable que almacena el número de intentos actuales.

```
#define INIT_TRANSACTION() \
  unsigned long __p_status, __p_retries
```

Código 4: Función INIT_TRANSACTION() del archivo transaction.h de la versión RTM-UniTransaccion.

BEGIN_TRANSACTION() : Función que intenta iniciar la transacción un número de veces almacenado en GLOBAL_RETRIES (Código 5). En caso de agotar los intentos serializa el código obteniendo el ticketlock. En cada reintento de actualiza la estructura de estadísticas utilizando profileAbortStatus(). Las variables declaradas en esta función son:

- **myticket** $\langle unsigned\ int \rangle$ **:** Variable utilizada almacenar el turno para obtener el lock.

```
#define BEGIN_TRANSACTION(thId, xId)                                                \
  __p_retries = 0;                                                                  \
  do                                                                                \
  {                                                                                 \
    if (__p_retries)                                                                \
      profileAbortStatus(__p_status, thId, xId);                                    \
    __p_retries++;                                                                  \
    if (__p_retries > GLOBAL_RETRIES)                                               \
    {  /* Funcion para actualizar el lock de forma atómica*/                        \
      unsigned int myticket = __sync_add_and_fetch(&(g_ticketlock.ticket), 1); \
      while (myticket != g_ticketlock.turn)                                         \
        CPU_RELAX();                                                                \
      break;                                                                        \
    }                                                                               \
    while (g_ticketlock.ticket >= g_ticketlock.turn)                                \
      CPU_RELAX(); /* Avoid Lemming effect */                                       \
} while ((__p_status = _xbegin())!= _XBEGIN_STARTED);
```

Código 5: Función BEGIN_TRANSACTION() del archivo transaction.h de la versión RTM-UniTransaccion.

COMMIT_TRANSACTION() : Función que finaliza la transacción en caso de ser exitosa, aborta en caso de que otro thread haya serializado el acceso a las variables compartidas o en caso de haber realizado las actualizaciones de las estructuras globales de manera serializada obteniendo el lock, se actualiza el lock (Código 6). En caso de que la transacción sea exitosa se actualiza la estructura de estadísticas utilizando profileCommit() y en caso de haber realizado las actualizaciones de manera serializada, se actualiza la estructura de estadísticas con profileFallback().

```
#define COMMIT_TRANSACTION(thId, xId)              \
  if (__p_retries <= GLOBAL_RETRIES)               \
  {                                                \
    if (g_ticketlock.ticket >= g_ticketlock.turn)  \
      _xabort(LOCK_TAKEN); /*Lazy subscription*/   \
    _xend();                                       \
    profileCommit(thId, xId, __p_retries - 1);     \
  }                                                \
  else                                             \
  {/* Funcion para actualizar el lock de forma atómica*/ \
    __sync_add_and_fetch(&(g_ticketlock.turn), 1); \
    profileFallback(thId, xId, __p_retries - 1);   \
  }
```

Código 6: Función COMMIT_TRANSACTION() del archivo transaction.h de la versión RTM-UniTransaccion.

profileAbortStatus() : Función que actualiza la estructura de estadísticas en función del motivo del aborto (Código 7). Como hemos comentado en las Secciones 2.2 y 2.2.1 Intel proporciona diferentes valores para entender el motivo del aborto.

```
// Funcion utilizada para contabilizar las transacciones que abortan. Intel recomienda
    que sea inline
static inline unsigned long profileAbortStatus(unsigned long eax, long thread, long xid)
{
  stats[thread][xid].xabortCount++;
  if (eax & _XABORT_EXPLICIT)
  {
    stats[thread][xid].explicitAborts++;
    if (_XABORT_CODE(eax) == LOCK_TAKEN)
      stats[thread][xid].explicitAbortsSubs++;
  }
  else if (eax & _XABORT_RETRY)
  {
    stats[thread][xid].retryAborts++;
    if (eax & _XABORT_CONFLICT)
      stats[thread][xid].retryConflictAborts++;
    if (eax & _XABORT_CAPACITY)
      stats[thread][xid].retryCapacityAborts++;
    if (eax & _XABORT_DEBUG)
      assert(0);
    if (eax & _XABORT_NESTED)
      assert(0);
  }
  else if (eax & _XABORT_CONFLICT)
  {
    stats[thread][xid].conflictAborts++;
  }
  else if (eax & _XABORT_CAPACITY)
  {
    stats[thread][xid].capacityAborts++;
  }
  else if (eax & _XABORT_DEBUG)
  {
    stats[thread][xid].debugAborts++;
  }
  else if (eax & _XABORT_NESTED)
  {
    stats[thread][xid].nestedAborts++;
  }
  else
  {
    //Todos los bits a cero (puede ocurrir por una llamada a CPUID u otra cosa)
    //Véase Section 8.3.5 RTM Abort Status Definition del Intel Architecture
    //Instruction Set Extensions Programming Reference (2012))
    stats[thread][xid].eaxzeroAborts++;
  }
  return 0;
}
```

Código 7: Función profileAbortStatus() del archivo transaction.h de la versión RTM-UniTransaccion.

profileCommit() : Función que actualiza la estructura de estadísticas para contabilizar las transacciones exitosas y el número de intentos (Código 8).

```
// Funcion utilizada para contabilizar las transacciones exitosas. Intel recomienda que
    sea inline
static inline void profileCommit(long thread, long xid, long retries)
{
  stats[thread][xid].xcommitCount++;
  stats[thread][xid].retryCCount += retries;
}
```

Código 8: Función profileCommit() del archivo transaction.h de la versión RTM-UniTransaccion.

profileFallback() : Función que actualiza la estructura de estadísticas para contabilizar las transacciones que han hecho fallback y han tenido que realizarse de manera no transaccional (Código 9).

```
// Funcion utilizada para contabilizar las transacciones que han hecho fallback y han
    tenido que realizarse de manera no transaccional. Intel recomienda que sea inline
static inline void profileFallback(long thread, long xid, long retries)
{
  stats[thread][xid].fallbackCount++;
  stats[thread][xid].retryFCount += retries;
}
```

Código 9: Función profileFallback() del archivo transaction.h de la versión RTM-UniTransaccion.

statsFileInit() : Función que inicializa la estructura de estadísticas y ticketlock utilizadas por los threads (Código 10).

dumpStats() : Función para volcar estadísticas a un fichero de salida (Código 10).

```
//Funciones para el fichero de estadísticas
int statsFileInit(char *file, long thCount, long xCount,long long int profileLength);
int dumpStats(char* tiempo,int memoria);
```

Código 10: Funciones statsFileInit() y dumpStats() del archivo transaction.h de la versión RTM-UniTransaccion.

3.1.4. transaction.c

En este archivo se inicializan las estructuras utilizadas para controlar el manejo de las transacciones RTM y se define una función para volcar las estadísticas almacenadas a un fichero de salida [1, 2, 4, 7, 19].

El archivo declara las siguientes variables globales:

- **fname** $\langle char[256] \rangle$ **:** Nombre del archivo a guardar archivos de salida.
- **threadCount** $\langle long \rangle$**:** Número de threads utilizados.
- **xactCount** $\langle long \rangle$**:** Número opcional para diferenciar estadísticas por cada thread.
- **stats** $\langle struct\ Stats\ ^{**} \rangle$**:** Estructura utilizada para almacenar las estadísticas.
- **g_ticketlock** $\langle struct\ TicketLock \rangle$**:** Ticketlock utilizado en caso de que sea necesario serializar la ejecución.

El archivo contiene las siguientes funciones:

statsFileInit() : Función que inicializa la estructura de estadísticas y ticketlock (Código 11).

```
// Funcion de inicializacion de estrucutura de estadisticas y ticketlock
int statsFileInit(char *file, long thCount, long xCount,long long int profileLength)
{
  int i, j;
  char ext[25];

  //Inicializo el ticket lock

    g_ticketlock.ticket=0;
    g_ticketlock.turn=1;


  strcpy(fname, file);

  //Inicio los arrays de estadísticas
  threadCount = thCount;
  xactCount = xCount;
  stats = (struct Stats **)malloc(sizeof(struct Stats *) * thCount);
  if (!stats)
    return 0;
  for (i = 0; i < thCount; i++)
  {
    stats[i] = (struct Stats *)malloc(sizeof(struct Stats) * xactCount);
    if (!stats[i])
      return 0;
  }

  for (i = 0; i < thCount; i++)
  {
    for (j = 0; j < xactCount; j++)
    {
      stats[i][j].xabortCount = 0;
      stats[i][j].explicitAborts = 0;
      stats[i][j].explicitAbortsSubs = 0;
      stats[i][j].retryAborts = 0;
      stats[i][j].retryCapacityAborts = 0;
      stats[i][j].retryConflictAborts = 0;
      stats[i][j].conflictAborts = 0;
      stats[i][j].capacityAborts = 0;
      stats[i][j].debugAborts = 0;
      stats[i][j].nestedAborts = 0;
      stats[i][j].eaxzeroAborts = 0;
      stats[i][j].xcommitCount = 0;
      stats[i][j].fallbackCount = 0;
      stats[i][j].retryCCount = 0;
      stats[i][j].retryFCount = 0;
    }
  }

  return 1;
}
```

Código 11: Función statsFileInit() del archivo transaction.c de la versión RTM-UniTransaccion.

dumpStats() : Función para volcar estadísticas a un fichero de salida (Código 12). Las variables declaradas en esta función son:

- **f** $\langle FILE\ ^{*} \rangle$ **:** Variable utilizada para escribir en el archivo con nombre almacenado en fname.
- **tmp** $\langle unsigned\ long\ int \rangle$ **:** Variable utilizada para contabilizar el número total de los datos guardados en la estructura de estadísticas.
- **comm** $\langle unsigned\ long\ int \rangle$ **:** Variable utilizada para contabilizar el número total de commits.
- **fall** $\langle unsigned\ long\ int \rangle$ **:** Variable utilizada para contabilizar el número total de fallbacks.
- **retComm** $\langle unsigned\ long\ int \rangle$ **:** Variable utilizada para contabilizar el número total de retries que han acabado en commit.

```
// Funcion para volcar estadisticas a un fichero de salida
int dumpStats(char* tiempo,int memoria)
{
  FILE *f;
  int i, j;
  unsigned long int tmp, comm, fall, retComm;

  //Creo el fichero
  f = fopen(fname, "w");
  if (!f)
    return 0;
  printf("Writing TM stats to: %s\n", fname);
  fprintf(f, "----------------------------------------\nOutput file: %s\n
    ---------------- Stats ----------------\n", fname);
  fprintf(f, "#Threads: %li\n", threadCount);
//-----------------------------------
  fprintf(f,"Time(ns): %s\n",tiempo);

  fprintf(f,"#Memory(B): %i\n",memoria);
//-----------------------------------
  fprintf(f, "Abort Count:");
  for (j = 0, tmp = 0; j < xactCount; j++)
  {
    fprintf(f, " XID%d: ", j);
    for (i = 0; i < threadCount; tmp += stats[i++][j].xabortCount)
      fprintf(f, "%lu ", stats[i][j].xabortCount);
  }
  fprintf(f, "Total: %lu\n", tmp);

  fprintf(f, "Explicit aborts:");
  for (j = 0, tmp = 0; j < xactCount; j++)
  {
    fprintf(f, " XID%d: ", j);
```

```
    for (i = 0; i < threadCount; tmp += stats[i++][j].explicitAborts)
      fprintf(f, "%lu ", stats[i][j].explicitAborts);
  }
  fprintf(f, " Total: %lu\n", tmp);

  fprintf(f, "  »     Subs:");
  for (j = 0, tmp = 0; j < xactCount; j++)
  {
    fprintf(f, " XID%d: ", j);
    for (i = 0; i < threadCount; tmp += stats[i++][j].explicitAbortsSubs)
      fprintf(f, "%lu ", stats[i][j].explicitAbortsSubs);
  }
  fprintf(f, " Total: %lu\n", tmp);

  fprintf(f, "Retry aborts:");
  for (j = 0, tmp = 0; j < xactCount; j++)
  {
    fprintf(f, " XID%d: ", j);
    for (i = 0; i < threadCount; tmp += stats[i++][j].retryAborts)
      fprintf(f, "%lu ", stats[i][j].retryAborts);
  }
  fprintf(f, " Total: %lu\n", tmp);

  fprintf(f, "  »     Con:");
  for (j = 0, tmp = 0; j < xactCount; j++)
  {
    fprintf(f, " XID%d: ", j);
    for (i = 0; i < threadCount; tmp += stats[i++][j].retryConflictAborts)
      fprintf(f, "%lu ", stats[i][j].retryConflictAborts);
  }
  fprintf(f, " Total: %lu\n", tmp);

  fprintf(f, "  »     Cap:");
  for (j = 0, tmp = 0; j < xactCount; j++)
  {
    fprintf(f, " XID%d: ", j);
    for (i = 0; i < threadCount; tmp += stats[i++][j].retryCapacityAborts)
      fprintf(f, "%lu ", stats[i][j].retryCapacityAborts);
  }
  fprintf(f, " Total: %lu\n", tmp);

  fprintf(f, "Conflict aborts:");
  for (j = 0, tmp = 0; j < xactCount; j++)
  {
    fprintf(f, " XID%d: ", j);
    for (i = 0; i < threadCount; tmp += stats[i++][j].conflictAborts)
      fprintf(f, "%lu ", stats[i][j].conflictAborts);
  }
  fprintf(f, " Total: %lu\n", tmp);

  fprintf(f, "Capacity aborts:");
  for (j = 0, tmp = 0; j < xactCount; j++)
  {
    fprintf(f, " XID%d: ", j);
    for (i = 0; i < threadCount; tmp += stats[i++][j].capacityAborts)
      fprintf(f, "%lu ", stats[i][j].capacityAborts);
  }
  fprintf(f, " Total: %lu\n", tmp);

  fprintf(f, "Debug aborts:");
  for (j = 0, tmp = 0; j < xactCount; j++)
  {
    fprintf(f, " XID%d: ", j);
```

```
    for (i = 0; i < threadCount; tmp += stats[i++][j].debugAborts)
      fprintf(f, "%lu ", stats[i][j].debugAborts);
  }
  fprintf(f, " Total: %lu\n", tmp);

  fprintf(f, "Nested aborts:");
  for (j = 0, tmp = 0; j < xactCount; j++)
  {
    fprintf(f, " XID%d: ", j);
    for (i = 0; i < threadCount; tmp += stats[i++][j].nestedAborts)
      fprintf(f, "%lu ", stats[i][j].nestedAborts);
  }
  fprintf(f, " Total: %lu\n", tmp);

  fprintf(f, "eax=0 aborts:");
  for (j = 0, tmp = 0; j < xactCount; j++)
  {
    fprintf(f, " XID%d: ", j);
    for (i = 0; i < threadCount; tmp += stats[i++][j].eaxzeroAborts)
      fprintf(f, "%lu ", stats[i][j].eaxzeroAborts);
  }
  fprintf(f, " Total: %lu\n", tmp);

  fprintf(f, "Commits:");
  for (j = 0, tmp = 0; j < xactCount; j++)
  {
    fprintf(f, " XID%d: ", j);
    for (i = 0; i < threadCount; tmp += stats[i++][j].xcommitCount)
      fprintf(f, "%lu ", stats[i][j].xcommitCount);
  }
  fprintf(f, " Total: %lu\n", tmp);
  comm = tmp;

  fprintf(f, "Fallbacks:");
  for (j = 0, tmp = 0; j < xactCount; j++)
  {
    fprintf(f, " XID%d: ", j);
    for (i = 0; i < threadCount; tmp += stats[i++][j].fallbackCount)
      fprintf(f, "%lu ", stats[i][j].fallbackCount);
  }
  fprintf(f, " Total: %lu\n", tmp);
  fall = tmp;

  fprintf(f, "RetriesCommited:");
  for (j = 0, tmp = 0; j < xactCount; j++)
  {
    fprintf(f, " XID%d: ", j);
    for (i = 0; i < threadCount; tmp += stats[i++][j].retryCCount)
      fprintf(f, "%lu ", stats[i][j].retryCCount);
  }
  fprintf(f, "Total: %lu ", tmp);
  retComm = tmp;
  fprintf(f, "PerXact: %f\n", (float)tmp / (float)comm);

  fprintf(f, "RetriesFallbacked:");
  for (j = 0, tmp = 0; j < xactCount; j++)
  {
    fprintf(f, " XID%d: ", j);
    for (i = 0; i < threadCount; tmp += stats[i++][j].retryFCount)
      fprintf(f, "%lu ", stats[i][j].retryFCount);
  }
  fprintf(f, "Total: %lu ", tmp);
  fprintf(f, "PerXact: %f\nRetriesAvg: %f\n", (float)tmp / (float)fall,
```

```
          (float)(retComm + tmp) / (float)(comm + fall));

  fclose(f);
  for (i = 0; i < threadCount; i++)
    free(stats[i]);
  free(stats);
  return 1;
}
}
```

Código 12: Función dumpStats() del archivo transaction.c de la versión RTM-UniTransaccion.

3.2. RTM-MultiTransaccion

En la versión RTM-UniTransaccion se realiza una única actualización (lectura y posible escritura) de las estructuras Matrix Profile y Matrix Profile index por cada transacción. En esta versión vamos a modificar el número de actualizaciones al número que se desee, almacenado por la variable tamTransaction introducida como entrada en tiempo de ejecución (Código 13).

Se diferencia a la versión anterior en la función SCRIMP() y SCAMP(), en la que se utiliza esta nueva variable tamTransaction.

```
''_
void scrimp(vector<DTYPE> &tSeries, vector<ITYPE> &idx, vector<DTYPE> &means, vector<
    DTYPE> &devs, vector<DTYPE> &profile, vector<ITYPE> &profileIndex)
{


#pragma omp parallel shared(profile,profileIndex)
  {
    INIT_TRANSACTION();
#pragma omp for schedule(dynamic)
    for (ITYPE ri = 0; ri < (ITYPE)idx.size(); ri++)
    {
      // Seleccion diagonal
      ITYPE diag = idx[ri];
      DTYPE dotProd = 0;

      // Calculo del primer dotProd de la diagonal
      for (ITYPE j = diag; j < windowSize + diag; j++)
        dotProd += tSeries[j] * tSeries[j - diag];

      // la i es la fila y j la columa de la matrix profile, cuya distancia se va a
    calcular
      ITYPE j = diag;
      ITYPE i = 0;

      // Calculo de la distancia
      DTYPE distance = 2 * (windowSize - (dotProd - windowSize * means[j] * means[i]) / (
    devs[j] * devs[i]));

      // Actualizacion del profile y profileIndex si la distancia actual es menor
      int myid=omp_get_thread_num();
      BEGIN_TRANSACTION(myid,0);
      if (distance < profile[j])
```

```
    {
      profile[j] = distance;
      profileIndex[j] = i;
    }

    if (distance < profile[i])
    {
      profile[i] = distance;
      profileIndex[i] = j;
    }
    COMMIT_TRANSACTION(myid,0);

    i = 1;
    ITYPE k=diag+1;

    // Actualizacion del profile y profileIndex si la distancia actual es menor
    for (ITYPE j = diag + 1; j < profileLength-tamTransaction+1; j+=tamTransaction)
    {

      BEGIN_TRANSACTION(myid,0);
      while(k<j+tamTransaction){

        dotProd += (tSeries[k + windowSize - 1] * tSeries[i + windowSize - 1]) - (
tSeries[k - 1] * tSeries[i - 1]);
        distance = 2 * (windowSize - (dotProd - means[k] * means[i] * windowSize) / (
devs[k] * devs[i]));


        if (distance < profile[k])
        {
          profile[k] = distance;
          profileIndex[k] = i;
        }

        if (distance < profile[i])
        {
          profile[i] = distance;
          profileIndex[i] = k;
        }


        i++;
        k++;
      }
      COMMIT_TRANSACTION(myid,0);
    }
    if(k<profileLength){
      BEGIN_TRANSACTION(myid,0);
      for(ITYPE l=k;l<profileLength;l++){

        dotProd += (tSeries[l + windowSize - 1] * tSeries[i + windowSize - 1]) - (
tSeries[l - 1] * tSeries[i - 1]);
        distance = 2 * (windowSize - (dotProd - means[l] * means[i] * windowSize) / (
devs[l] * devs[i]));


        if (distance < profile[l])
        {
          profile[l] = distance;
          profileIndex[l] = i;
        }

        if (distance < profile[i])
```

```
          {
            profile[i] = distance;
            profileIndex[i] = l;
          }


          i++;
        }

        COMMIT_TRANSACTION(myid,0);
      }

    } //'pragma omp for' asegura que a partir de aqui todos los threads han acabado


  }
}
```

Código 13: Función SCRIMP() del archivo SCRIMP.cpp de la versión RTM-MultiTransaccion.

3.3. RTM-DiagTransaccion

En esta versión se propone calcular todas las distancias de la diagonal, almacenarlas todas y una vez calculadas actualizar las estructuras globales a partir de los profile privados utilizados por los threads protegidos por la transacción (Código 14). Es decir que el número de actualizaciones por transacción es igual al tamaño de la diagonal. En concreto se ejecutan $diagonal = profilength - exclusionZone$ actualizaciones como máximo.

```
''_
void scrimp(vector<DTYPE> &tSeries, vector<ITYPE> &idx, vector<DTYPE> &means, vector<
    DTYPE> &devs, vector<DTYPE> &profile, vector<ITYPE> &profileIndex)
{

  // Buffer privado utilizado por los Threads
  vector<DTYPE> buffer(profileLength * numThreads);

#pragma omp parallel shared(profile,profileIndex)
  {
    // Suponiendo que ITYPE como uint32_t (podemos indexar series de maximo 2^32
    elementos o 4G elementos), pero para indexar profile_tmp, que es num_threads
    // mas grande que profile necesitaremos mas bits por lo que usamos uint64_t. Ejemplo
    600000 elementos y 8 threads.
    uint64_t my_offset = omp_get_thread_num() * profileLength;

    INIT_TRANSACTION();
#pragma omp for schedule(dynamic)
    for (ITYPE ri = 0; ri < (ITYPE)idx.size(); ri++)
    {
      // Seleccion diagonal
      ITYPE diag = idx[ri];
      DTYPE dotProd = 0;
```

```
    // Calculo del primer dotProd de la diagonal
    for (ITYPE j = diag; j < windowSize + diag; j++)
      dotProd += tSeries[j] * tSeries[j - diag];

    // i es la fila y j la columna de la matrix profile, cuya distancia se va a
  calcular
    ITYPE j = diag;
    ITYPE i = 0;

    // Calculo de la distancia
    DTYPE distance = 2 * (windowSize - (dotProd - windowSize * means[j] * means[i]) / (
  devs[j] * devs[i]));

    // Actualizacion del profile y profileIndex si la distancia actual es menor
    int myid=omp_get_thread_num();
    BEGIN_TRANSACTION(myid,0);
    if (distance < profile[j])
    {
      profile[j] = distance;
      profileIndex[j] = i;
    }

    if (distance < profile[i])
    {
      profile[i] = distance;
      profileIndex[i] = j;
    }
    COMMIT_TRANSACTION(myid,0);

    i = 1;
    j++;

    // Calculo de las distancias previo a la transaccion
    int resto=profileLength-j;
    for(ITYPE k=0;k<resto;k++){
      dotProd += (tSeries[j + windowSize - 1] * tSeries[i + windowSize - 1]) - (tSeries
  [j - 1] * tSeries[i - 1]);
      distance = 2 * (windowSize - (dotProd - means[j] * means[i] * windowSize) / (devs
  [j] * devs[i]));

      buffer[my_offset+k]=distance;
      i++;
      j++;
    }
    //Actualizacion del profile y profileIndex si la distancia actual es menor
    BEGIN_TRANSACTION(myid,0);
    for(ITYPE k=0;k<resto;k++){

      if (buffer[my_offset+k] < profile[j-resto+k])
      {
        profile[j-resto+k] = buffer[my_offset+k];
        profileIndex[j-resto+k] = i-resto+k;
      }

      if (buffer[my_offset+k] < profile[i-resto+k])
      {
        profile[i-resto+k] = buffer[my_offset+k];
        profileIndex[i-resto+k] = j-resto+k;
      }
    }
    COMMIT_TRANSACTION(myid,0);
```

```
    } //'pragma omp for' asegura que a partir de aqui todos los threads han acabado

  }
}
```

Código 14: Función SCRIMP() del archivo SCRIMP.cpp de la versión RTM-DiagTransaccion.

3.4. RTM-DinamicTransaccion

En la versión RTM-DiagTransaccion se calculan todas las distancias de la diagonal, se almacenan y una vez calculadas se actualizan las estructuras globales. En esta versión en vez de calcular todas las distancias de la diagonal, se calculan un número de distancias de la diagonal que se desee y a continuación se actualizan las estructuras globales (Código 15). El número de distancias es almacenado por la variable tamTransaction introducida como entrada en tiempo de ejecución.

Se diferencia de la versión anterior en la función SCRIMP() y SCAMP(), en la que se utiliza esta nueva variable tamTransaction.

```
''_
void scrimp(vector<DTYPE> &tSeries, vector<ITYPE> &idx, vector<DTYPE> &means, vector<
    DTYPE> &devs, vector<DTYPE> &profile, vector<ITYPE> &profileIndex)
{

  // Buffer privado utilizado por los Threads
  vector<DTYPE> buffer(tamTransaction * numThreads);

#pragma omp parallel shared(profile,profileIndex)
  {
    // Suponiendo que ITYPE como uint32_t (podemos indexar series de maximo 2^32
    elementos o 4G elementos), pero para indexar profile_tmp, que es num_threads
    // mas grande que profile necesitaremos mas bits por lo que usamos uint64_t. Ejemplo
    600000 elementos y 8 threads.
    uint64_t my_offset = omp_get_thread_num() * tamTransaction;

    INIT_TRANSACTION();
#pragma omp for schedule(dynamic)
    for (ITYPE ri = 0; ri < (ITYPE)idx.size(); ri++)
    {
      // Seleccion diagonal
      ITYPE diag = idx[ri];
      DTYPE dotProd = 0;

      // Calculo del primer dotProd de la diagonal
      for (ITYPE j = diag; j < windowSize + diag; j++)
        dotProd += tSeries[j] * tSeries[j - diag];

      // i es la fila y j la columna de la matrix profile, cuya distancia se va a
    calcular
      ITYPE j = diag;
      ITYPE i = 0;
```

```
    // Calculo de la distancia
    DTYPE distance = 2 * (windowSize - (dotProd - windowSize * means[j] * means[i]) / (
  devs[j] * devs[i]));

    // Actualizacion del profile y profileIndex si la distancia actual es menor
    int myid=omp_get_thread_num();
    BEGIN_TRANSACTION(myid,0);
    if (distance < profile[j])
    {
      profile[j] = distance;
      profileIndex[j] = i;
    }

    if (distance < profile[i])
    {
      profile[i] = distance;
      profileIndex[i] = j;
    }
    COMMIT_TRANSACTION(myid,0);

    i = 1;
    j++;

    // Calculo del resto de distancias en la diagonal

    while(j<profileLength-tamTransaction+1){
      // Calculo de las distancias previo a la transaccion
      for(ITYPE k=0;k<tamTransaction;k++){
        dotProd += (tSeries[j + windowSize - 1] * tSeries[i + windowSize - 1]) - (
  tSeries[j - 1] * tSeries[i - 1]);
        distance = 2 * (windowSize - (dotProd - means[j] * means[i] * windowSize) / (
  devs[j] * devs[i]));

        buffer[my_offset+k]=distance;
        i++;
        j++;
      }

      //Actualizacion del profile y profileIndex si la distancia actual es menor
      BEGIN_TRANSACTION(myid,0);
      for(ITYPE k=0;k<tamTransaction;k++){

        if (buffer[my_offset+k] < profile[j-tamTransaction+k])
        {
          profile[j-tamTransaction+k] = buffer[my_offset+k];
          profileIndex[j-tamTransaction+k] = i-tamTransaction+k;
        }

        if (buffer[my_offset+k] < profile[i-tamTransaction+k])
        {
          profile[i-tamTransaction+k] = buffer[my_offset+k];
          profileIndex[i-tamTransaction+k] = j-tamTransaction+k;
        }
      }
      COMMIT_TRANSACTION(myid,0);

    }

    if(j<profileLength){
      // Calculo de las distancias previo a la transaccion
      int resto=profileLength-j;
      for(ITYPE k=0;k<resto;k++){
```

```
          dotProd += (tSeries[j + windowSize - 1] * tSeries[i + windowSize - 1]) - (
    tSeries[j - 1] * tSeries[i - 1]);
          distance = 2 * (windowSize - (dotProd - means[j] * means[i] * windowSize) / (
    devs[j] * devs[i]));

          buffer[my_offset+k]=distance;
          i++;
          j++;
        }
        //Actualizacion del profile y profileIndex si la distancia actual es menor
        BEGIN_TRANSACTION(myid,0);
        for(ITYPE k=0;k<resto;k++){

          if (buffer[my_offset+k] < profile[j-resto+k])
          {
            profile[j-resto+k] = buffer[my_offset+k];
            profileIndex[j-resto+k] = i-resto+k;
          }

          if (buffer[my_offset+k] < profile[i-resto+k])
          {
            profile[i-resto+k] = buffer[my_offset+k];
            profileIndex[i-resto+k] = j-resto+k;
          }
        }
        COMMIT_TRANSACTION(myid,0);
      }

    } //'pragma omp for' asegura que a partir de aqui todos los threads han acabado


  }

}
```

Código 15: Función SCRIMP() del archivo SCRIMP.cpp de la versión RTM-DinamicTransaccion.

3.5. OpenMP-SingleLock

En esta versión se propone serializar totalmente el acceso a las estructuras globales utilizando OpenMP como lock (Código 16) [5]. Por lo que no se utiliza RTM, ni los archivos transaction.c y transaction.h.

```
''_
void scrimp(vector<DTYPE> &tSeries, vector<ITYPE> &idx, vector<DTYPE> &means, vector<
    DTYPE> &devs, vector<DTYPE> &profile, vector<ITYPE> &profileIndex)
{


#pragma omp parallel shared(profile,profileIndex)
  {

#pragma omp for schedule(dynamic)
    for (ITYPE ri = 0; ri < (ITYPE)idx.size(); ri++)
    {
```

```
    // Seleccion diagonal
    ITYPE diag = idx[ri];
    DTYPE dotProd = 0;

    // Calculo del primer dotProd de la diagonal
    for (ITYPE j = diag; j < windowSize + diag; j++)
      dotProd += tSeries[j] * tSeries[j - diag];

    // la i es la fila y j la columa de la matrix profile, cuya distancia se va a
  calcular
    ITYPE j = diag;
    ITYPE i = 0;

    // Calculo de la distancia
    DTYPE distance = 2 * (windowSize - (dotProd - windowSize * means[j] * means[i]) / (
  devs[j] * devs[i]));

    // Actualizacion del profile y profileIndex si la distancia actual es menor
    #pragma omp critical
    {
      if (distance < profile[j])
      {
        profile[j] = distance;
        profileIndex[j] = i;
      }
    }
    #pragma omp critical
    {
      if (distance < profile[i])
      {
        profile[i] = distance;
        profileIndex[i] = j;
      }
    }


    i = 1;

    // Calculo del resto de distancias en la diagonal
    for (ITYPE j = diag + 1; j < profileLength; j++)
    {
      dotProd += (tSeries[j + windowSize - 1] * tSeries[i + windowSize - 1]) - (tSeries
  [j - 1] * tSeries[i - 1]);
      distance = 2 * (windowSize - (dotProd - means[j] * means[i] * windowSize) / (devs
  [j] * devs[i]));

      #pragma omp critical
      {
        if (distance < profile[j])
        {
          profile[j] = distance;
          profileIndex[j] = i;
        }
      }
      #pragma omp critical
      {
        if (distance < profile[i])
        {
          profile[i] = distance;
          profileIndex[i] = j;
        }
      }
```

```
        i++;
      }
    } //'pragma omp for' asegura que a partir de aqui todos los threads han acabado


  }
}
```

Código 16: Función SCRIMP() del archivo SCRIMP.cpp de la versión OpenMP-SingleLock.

3.6. OpenMP-MultiLock

En la versión OpenMP-SingleLock utilizamos un único lock para serializar el acceso a las estructuras globales. En esta versión se propone utilizar locks de grano fino (Código 17), utilizando un array de locks que serializan el acceso de cada posición de las estructuras globales[5]. Por lo que no se utiliza RTM, ni los archivos transaction.c y transaction.h.

```
''_
void scrimp(vector<DTYPE> &tSeries, vector<ITYPE> &idx, vector<DTYPE> &means, vector<
    DTYPE> &devs, vector<DTYPE> &profile, vector<ITYPE> &profileIndex)
{

  // Creacion e inicializacion de la estructura de locks
  omp_lock_t *locks=new omp_lock_t[profileLength];
  for(int i=0;i<profileLength;i++){
    omp_init_lock(&(locks[i]));
  }

#pragma omp parallel shared(profile,profileIndex,locks)
  {

#pragma omp for schedule(dynamic)
    for (ITYPE ri = 0; ri < (ITYPE)idx.size(); ri++)
    {
      // Seleccion diagonal
      ITYPE diag = idx[ri];
      DTYPE dotProd = 0;

      // Calculo del primer dotProd de la diagonal
      for (ITYPE j = diag; j < windowSize + diag; j++)
        dotProd += tSeries[j] * tSeries[j - diag];

      // la i es la fila y j la columa de la matrix profile, cuya distancia se va a
    calcular
      ITYPE j = diag;
      ITYPE i = 0;

      // Calculo de la distancia
      DTYPE distance = 2 * (windowSize - (dotProd - windowSize * means[j] * means[i]) / (
    devs[j] * devs[i]));

      // Actualizacion del profile y profileIndex si la distancia actual es menor
      omp_set_lock(&(locks[j]));
      if (distance < profile[j])
      {
        profile[j] = distance;
```

```
        profileIndex[j] = i;
      }
      omp_unset_lock(&(locks[j]));

      omp_set_lock(&(locks[i]));
      if (distance < profile[i])
      {
        profile[i] = distance;
        profileIndex[i] = j;
      }
      omp_unset_lock(&(locks[i]));

      i = 1;

      // Calculo del resto de distancias en la diagonal
      for (ITYPE j = diag + 1; j < profileLength; j++)
      {
        dotProd += (tSeries[j + windowSize - 1] * tSeries[i + windowSize - 1]) - (tSeries
    [j - 1] * tSeries[i - 1]);
        distance = 2 * (windowSize - (dotProd - means[j] * means[i] * windowSize) / (devs
    [j] * devs[i]));

        omp_set_lock(&(locks[j]));
        if (distance < profile[j])
        {
          profile[j] = distance;
          profileIndex[j] = i;
        }
        omp_unset_lock(&(locks[j]));

        omp_set_lock(&(locks[i]));
        if (distance < profile[i])
        {
          profile[i] = distance;
          profileIndex[i] = j;
        }
        omp_unset_lock(&(locks[i]));

        i++;
      }
    } //'pragma omp for' asegura que a partir de aqui todos los threads han acabado


  }
  delete locks;
}
```

Código 17: Función SCRIMP() del archivo SCRIMP.cpp de la versión OpenMP-MultiLock.

3.7. Versión privatizada

En esta versión se utiliza paralelismo donde existe las estructuras globales Matrix Profile y Matrix Profile index pero no se comparten al mismo tiempo. Sino que cada thread tiene su propio Matrix Profile y Matrix Profile index privados, donde calculan y almacenan sus valores locales, y al terminar se realiza una etapa de reducción final actualizando el Matrix Profile y

Matrix Profile index global (Código 18). Al no haber necesidad de sincronización que limite el acceso de los threads a las estructuras globales, no es necesario utilizar RTM o locks en esta versión.

Es la opción más utilizada ya que se obtienen los mejores tiempos de ejecución, aunque tiene el problema de la necesidad de utilizar mucha más memoria. Siendo n el número de threads utilizados, se utilizan n Matrix Profile y Matrix Profile index, además del Matrix Profile y Matrix Profile index global. También, al no tener la necesidad de sincronizar ninguna estructura global compartida por los threads, es una versión menos compleja de programar.

```
''_
void scrimp(vector<DTYPE> &tSeries, vector<ITYPE> &idx, vector<DTYPE> &means, vector<
    DTYPE> &devs, vector<DTYPE> &profile, vector<ITYPE> &profileIndex)
{
  // Estructuras privadas utilizadas por los Threads
  vector<DTYPE> profile_tmp(profileLength * numThreads);
  vector<ITYPE> profileIndex_tmp(profileLength * numThreads);

#pragma omp parallel
  {
    /* Suponiendo que ITYPE como uint32_t (podemos indexar series de maximo 2^32
    elementos o 4G elementos), pero para indexar profile_tmp, que es num_threads
     mas grande que profile necesitaremos mas bits por lo que usamos uint64_t. Ejemplo
    600000 elementos y 8 threads*/
    uint64_t my_offset = omp_get_thread_num() * profileLength;

    // Inicializacion del profile privado de cada thread
    for (uint64_t i = my_offset; i < (my_offset + profileLength); i++)
      profile_tmp[i] = numeric_limits<DTYPE>::infinity();

#pragma omp for schedule(dynamic)
    for (ITYPE ri = 0; ri < (ITYPE)idx.size(); ri++){

      // Seleccion diagonal
      ITYPE diag = idx[ri];
      DTYPE dotProd = 0;

      // Calculo del primer dotProd de la diagonal
      for (ITYPE j = diag; j < windowSize + diag; j++)
        dotProd += tSeries[j] * tSeries[j - diag];

      // i es la fila y j la columna de la matrix profile, cuya distancia se va a
    calcular
      ITYPE j = diag;
      ITYPE i = 0;

      // Calculo de la distancia
      DTYPE distance = 2 * (windowSize - (dotProd - windowSize * means[j] * means[i]) / (
    devs[j] * devs[i]));

      // Actualizacion del profile y profileIndex si la distancia actual es menor
      if (distance < profile_tmp[my_offset + j])
      {
        profile_tmp[my_offset + j] = distance;
        profileIndex_tmp[my_offset + j] = i;
      }
```

```
      if (distance < profile_tmp[my_offset + i])
      {
        profile_tmp[my_offset + i] = distance;
        profileIndex_tmp[my_offset + i] = j;
      }

      i = 1;

      // Calculo del resto de distancias en la diagonal
      for (ITYPE j = diag + 1; j < profileLength; j++)
      {
        dotProd += (tSeries[j + windowSize - 1] * tSeries[i + windowSize - 1]) - (tSeries
    [j - 1] * tSeries[i - 1]);
        distance = 2 * (windowSize - (dotProd - means[j] * means[i] * windowSize) / (devs
    [j] * devs[i]));

        if (distance < profile_tmp[my_offset + j])
        {
          profile_tmp[my_offset + j] = distance;
          profileIndex_tmp[my_offset + j] = i;
        }

        if (distance < profile_tmp[my_offset + i])
        {
          profile_tmp[my_offset + i] = distance;
          profileIndex_tmp[my_offset + i] = j;
        }
        i++;
      }
    } //'pragma omp for' asegura que a partir de aqui todos los threads han acabado

    DTYPE min_distance;
    ITYPE min_index;
// Actualizacion del profile global a partir de los profile privados utilizados por los
    threads
#pragma omp for schedule(static)
    for (ITYPE colum = 0; colum < profileLength; colum++)
    {
      min_distance = numeric_limits<DTYPE>::infinity();
      min_index = 0;
      for (ITYPE th = 0; th < numThreads; th++)
      {
        if (profile_tmp[colum + (th * profileLength)] < min_distance)
        {
          min_distance = profile_tmp[colum + (th * profileLength)];
          min_index = profileIndex_tmp[colum + (th * profileLength)];
        }
      }
      profile[colum] = min_distance;
      profileIndex[colum] = min_index;
    }
  }
}
```

Código 18: Función SCRIMP() del archivo SCRIMP.cpp de la versión privatizada.

4

Experimentación y resultados

En esta sección se describe como se han obtenido los resultados obtenidos de la experimentación de las diferentes versiones. Destacar que se ha optado por el uso del lenguaje de programación Python para representar las gráficas.

En primer lugar, se explica como se ha llevado a cabo la experimentación, seguido de un análisis de sensibilidad del tamaño de la transacción de las versiones RTM-MultiTransaccion y RTM-DinamicTransaccion, se muestran los resultados obtenidos y el consumo de memoria, y se finaliza con una discusión sobre los resultados.

4.1. Metodología experimental

Para llevar a cabo las pruebas necesarias de las diferentes versiones, se han utilizado las series temporales [6] detalladas en la Fig. 3, donde se muestra el tamaño de ventana (o tamaño de la subsecuencia, m), el número de muestras (n), el mínimo y el máximo valor.

Todos los experimentos se han realizado utilizando el servidor xeongold detallado en la Sección 1.4.5 [3]. En el Apéndice A se muestran las diferentes instrucciones utilizadas para la ejecución de los experimentos.

Primero se ha realizado un análisis de sensibilidad de las versiones RTM-MultiTransaccion y RTM-DinamicTransaccion. Una vez realizado el análisis, se ha escogido un tamaño de transacción adecuado para los experimentos.

Una vez escogido los tamaños de transacción para las versiones RTM-MultiTransaccion y RTM-DinamicTransaccion, se ha realizado diferentes ejecuciones de todas las versiones, utilizando las series temporales descritas en la Fig. 3, calculando los tiempos de ejecución mientras se aumenta el número de threads hasta 64 threads. Destacar que las series temporales más

grandes (1.800.000 datos) se han utilizado sólo para comparar la mejor versión RTM (RTM-MultiTransaccion) con la versión Privado.

Al ser la versión Privado, la opción más utilizada, se tomará su tiempo de ejecución como base para calcular el speed-up del resto de versiones.

Time Series	n	m	Max	Min
Audio	20.234	200	6,69	-56,48
ECG	180.000	500	2,6	0,32
Power	180.000	1.325	140	0
Seismology	180.000	50	696,08	-185,72
Human act.	7.997	120	2,5	-1,89
Peng. Behav.	109.842	800	0.51	-0,2
ECG	1.800.000	500	3,39	-1,635
Power	1.859.587	1.325	140	0
Seismology	1.728.000	50	2329,3	-2334,6

Figura 3: Series Temporales.

4.2. Análisis de sensibilidad del tamaño de transacción

Las Figuras 4 y 5 muestran como impacta el modificar el tamaño de la transacción utilizando un thread y las series temporales (Fig. 3) para las versiones RTM-MultiTransaccion y RTM-DinamicTransaccion.

Se ha calculado los tiempos de ejecución aumentando el tamaño de la transacción (1 actualización hasta 524288 actualizaciones en potencias de 2).

Para entender mejor las diferentes gráficas se puede observar porque abortan las transacciones. La Figura 6 muestra un ejemplo de los diferentes tipos de abortos obtenidos en la ejecución del análisis. Se observa que predomina los abortos por capacidad. Ésto sucede porque los abortos explícitos y por conflictos son debidos a la interacción entre diferentes threads. Al realizar el análisis con un thread, éstos no ocurren. Por lo que se va a tener en cuenta los abortos por capacidad.

Las Figuras 7 y 8 muestran los abortos por capacidad generados al aumentar el tamaño

de transacción. El máximo valor, en la versión RTM-MultiTransaccion, nos indica el mínimo tamaño de transacción que produce un aborto de transacción. Al ser el mínimo tamaño de transacción, la división de los cálculos de la diagonal se realiza en el máximo de divisiones posibles. Por esta razón genera el número máximo de abortos por capacidad. Los abortos por capacidad ”disminuyen” a partir de este valor. Pero ésto se debe a que al ser el tamaño de transacción mayor, el número de divisiones es menor, por lo que aparenta que son pocos. En realidad el porcentaje de que una transacción aborte no deja de incrementar. Ésto se puede observar en la Figura 9, que es un ejemplo de la probabilidad que tiene una transacción de abortar. Este máximo valor no está presente en la versión RTM-DinamicTransaccion porque el tamaño de transacción no llega a ser lo suficientemente grande para generar el aborto de transacción. Sería posible conseguir el tamaño indicado utilizando series temporales más grandes, lo que permitiría utilizar tamaños de transacciones más grandes. A partir de ahora utilizaremos la versión RTM-DiagTransaccion que utiliza la diagonal entera como transacción utilizando ésta, el tamaño máximo de transacción de la versión RTM-DinamicTransaccion.

En las Figuras 4 y 5 se observa que en este tamaño de transacción mínimo que genera un aborto, se consigue un tiempo de ejecución peor. A partir de este valor las gráficas muestran que el tiempo de ejecución mejora al aumentar el tamaño de transacción, pero no es realmente cierto, lo que ocurre es que al aumentar el tamaño de transacción, el número de divisiones de la diagonal es menor, por lo que existen menos transacciones. Éstas transacciones tienen un alto porcentaje de abortar y al ser menos transacciones, acaba serializándose la ejecución del programa y al hacer el análisis con un thread, serializar la ejecución del algoritmo nos genera un buen tiempo de ejecución. Ésto no sucedería utilizando más threads.

Por estas razones, se debe observar las gráficas desde el comienzo hasta el tamaño de transacción mínimo que genera un aborto. En la versión RTM-MultiTransacción, el valor es 512.

Ahora, para obtener el tamaño de transacción óptimo utilizando un thread, se debe observar en las Figuras 4 y 5 desde el comienzo hasta 512, en la versión RTM-MultiTransacción.

En la versión RTM-MultiTransacción el tamaño de transacción óptimo para un thread es 128. Los procesadores del servidor xeongold permiten utilizar HyperThreading por lo que los recursos que disponen las CPUs se comparten entre dos CPUs. Hay que considerar ésto al realizar la ejecución utilizando 64 threads por lo que se disminuirá el tamaño de transacción a

un valor menor (64). Además al aumentar el número de threads, el poseer tamaños de transacción grandes aumenta la probabilidad de conflictos entre éstas. También debemos tener ésto en cuenta diminuyendo el valor a 16.

Intel nos informa de que al utilizar transacciones muy pequeñas, el procesador pasa gran parte del tiempo actualizando la memoria y la CPU tendría poca carga de trabajo y si las transacciones son grandes, existen más posibilidades de que la transacción aborte [15][16]. Por lo que habria que buscar un termino intermedio de tamaño de la transacción.

Se observa que son las mismas conclusiones que se pueden obtener de las ejecuciones.

Se concluye diciendo que el tamaño de transacción óptimo a utilizar en el resto de la experimentación es 16 para RTM-MultiTransacción y se utilizará la versión RTM-DiagTransaccion en vez de la versión RTM-DinamicTransaccion.

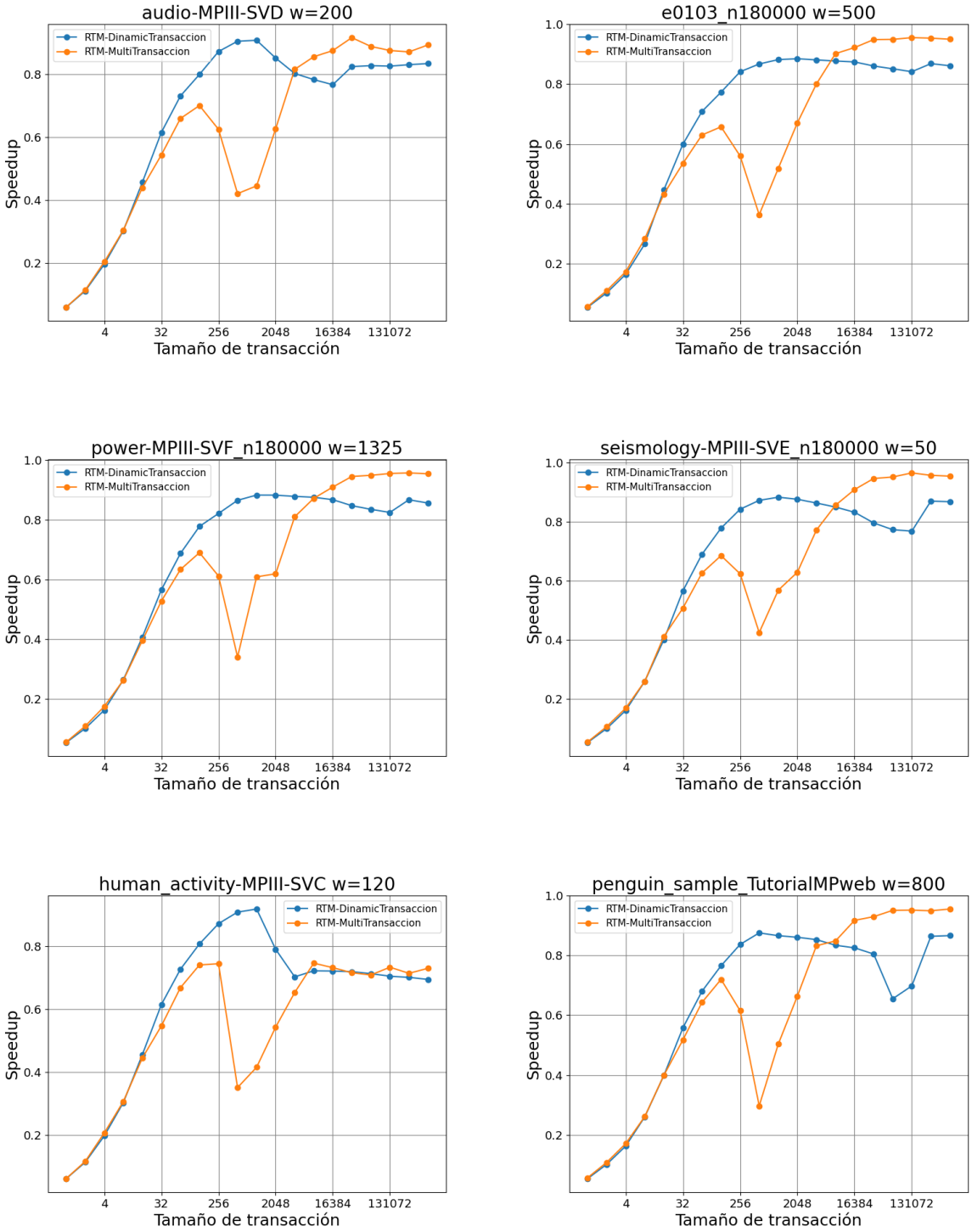

Figura 4: Análisis de sensibilidad del tamaño de transacción utilizando SCRIMP.

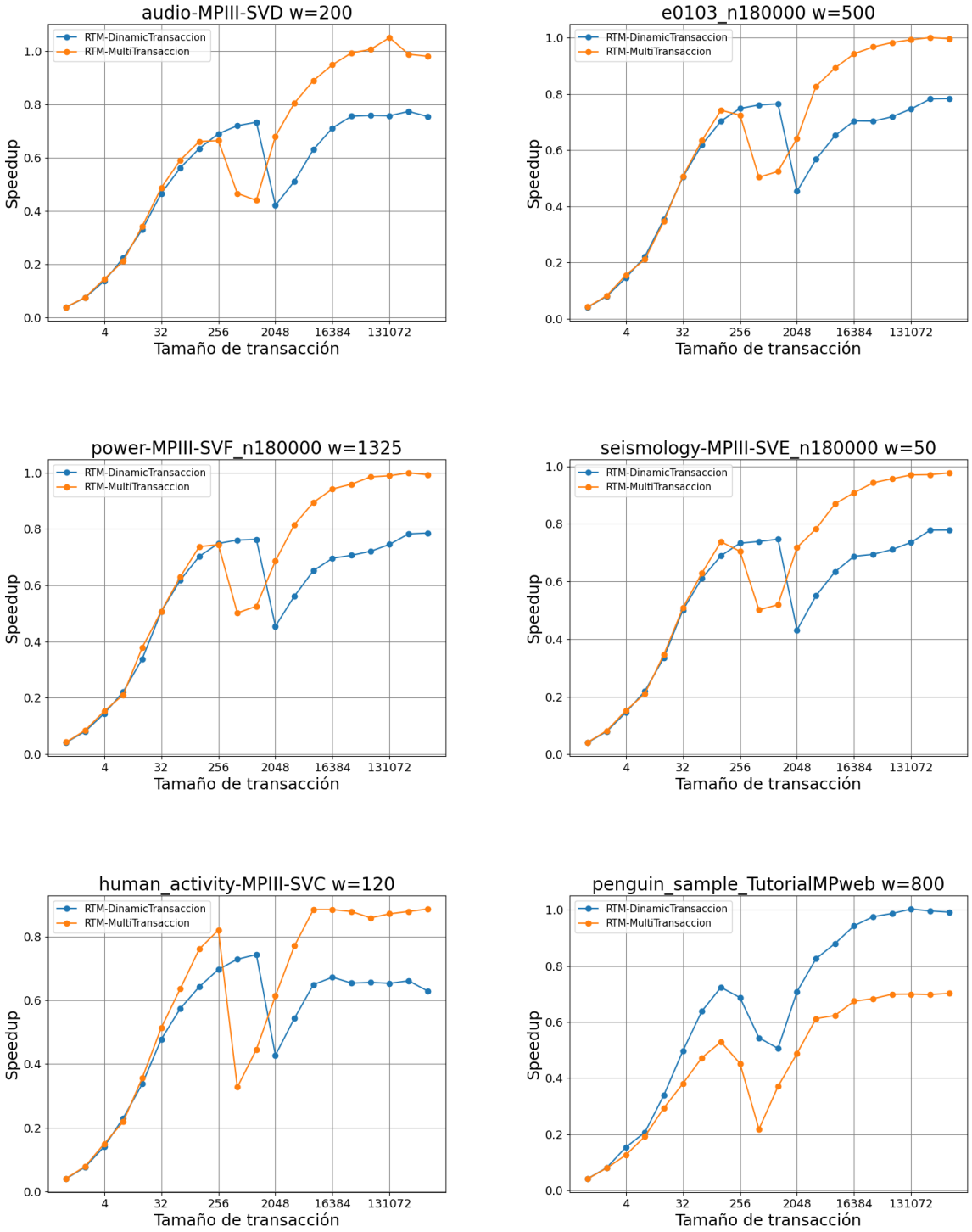

Figura 5: Análisis de sensibilidad del tamaño de transacción utilizando SCAMP.

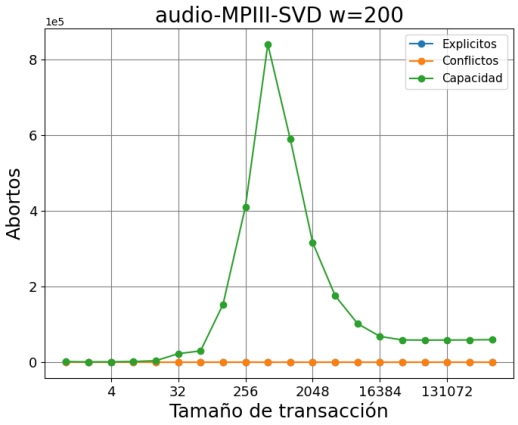

Figura 6: Ejemplo de abortos utilizando SCRIMP.

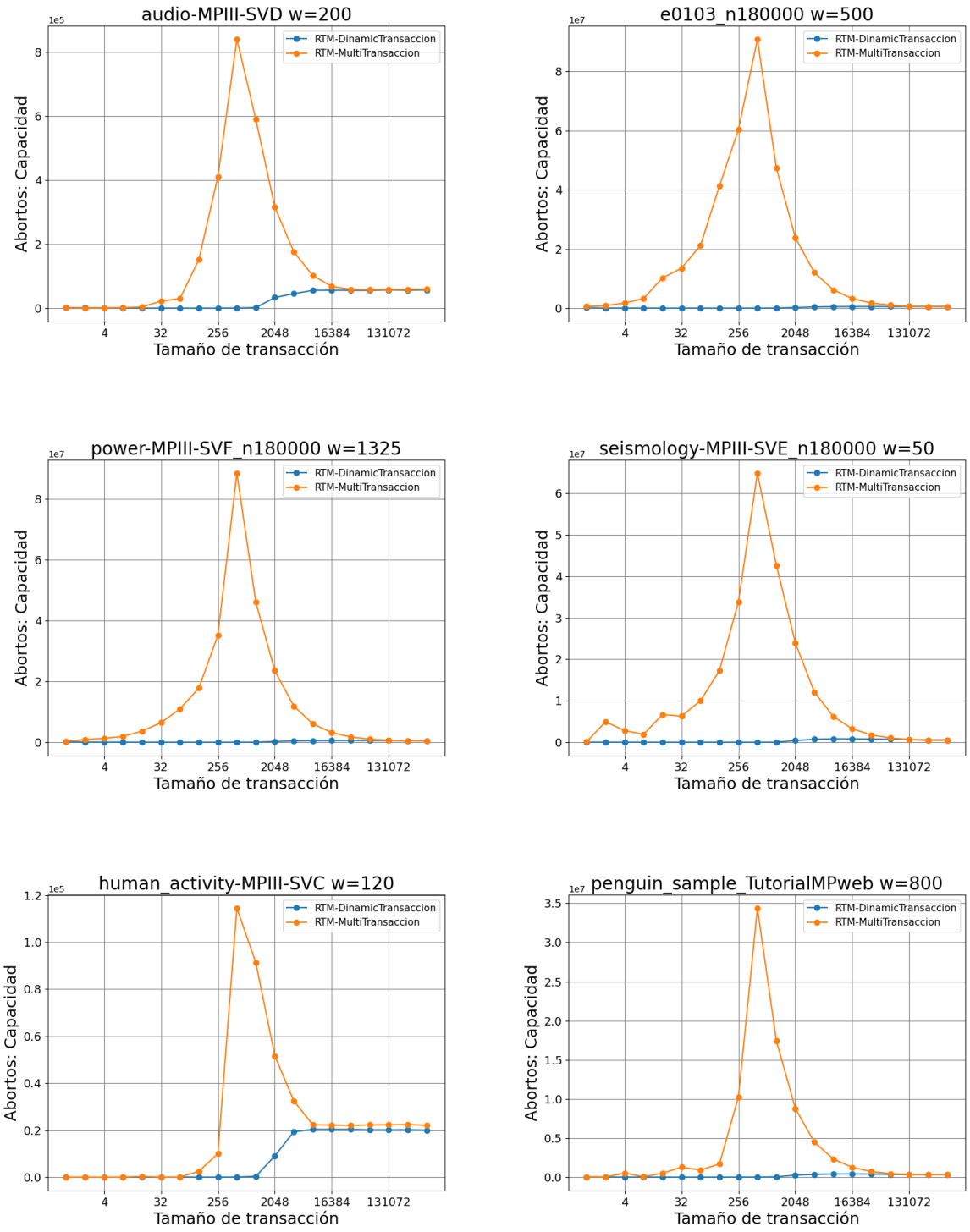

Figura 7: Abortos por capacidad aumentando el tamaño de transacción utilizando SCRIMP.

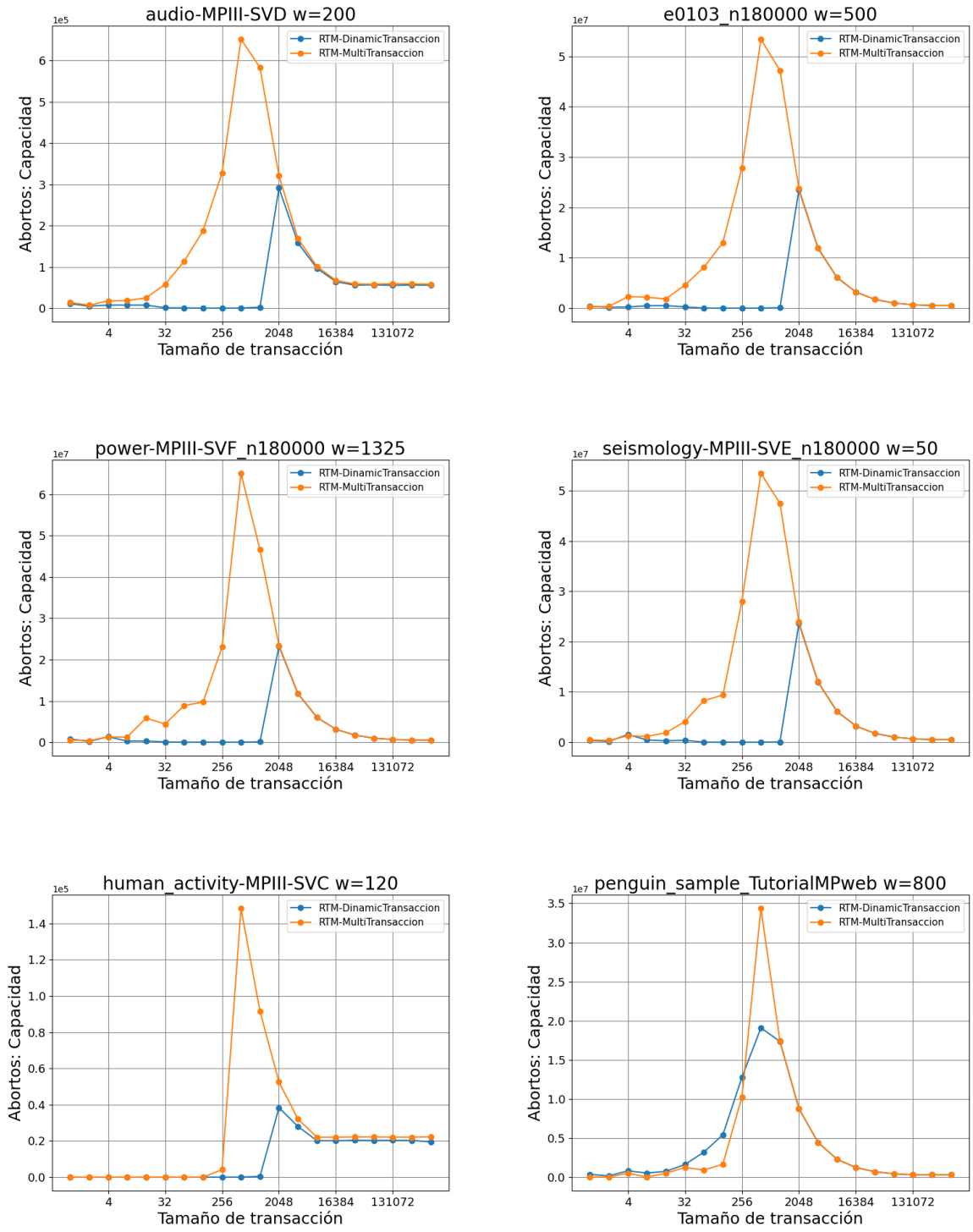

Figura 8: Abortos por capacidad aumentando el tamaño de transacción utilizando SCAMP.

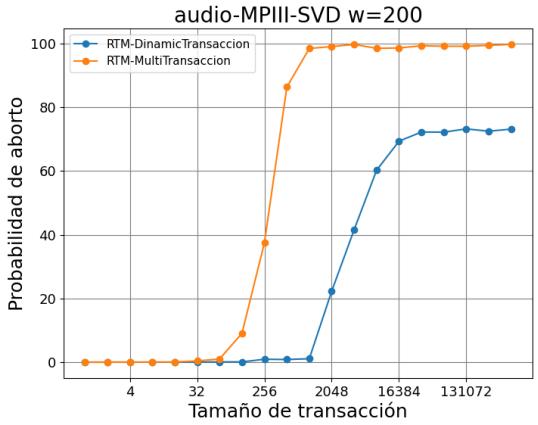

Figura 9: Ejemplo de probabilidad de que una transacción aborte utilizando SCRIMP.

4.3. Rendimiento

Las Figuras 10 y 11 muestran los speed-up obtenidos de las diferentes versiones aumentando el número de threads desde 1 hasta 64. Se han utilizado las series temporales descritas en la Figura 3. La versión OpenMP-SingleLock al tener tiempos de ejecución muy largos, sólo se han calculado los tiempos de ejecución con las series temporales de menor tamaño (Audio y Human Activity). Además como hemos comentando en la Sección 4.2 se utilizará la versión RTM-DiagTransacción en vez de la versión RTM-DinamicTransacción y se ha optado por no representar la versión RTM-UniTransacción porque al tener un tamaño de transacción muy pequeño, los tiempos de ejecución son largos debido a las barreras de memoria y además la versión RTM-MultiTransaccion es una versión mejorada de ésta.

Se puede observar que la versión Privado es la mejor versión de todas en cuanto tiempos de ejecución. En segundo lugar, la versión RTM-MultiTransaccion con un tiempo de ejecución un poco mayor. Seguido de la versión RTM-DiagTransaccion. Luego la versión OpenMP-MultiLock. Y por último la versión OpenMP-SingleLock. Hay que tener en cuenta que algunas versiones utilizan menos memoria que otras.

Los resultados muestran, en la versión OpenMP-SingleLock, por qué era un paradigma el hecho de utilizar estructuras globales utilizando paralelismo y sincronización de manera convencional a través de los locks. Los threads al tener que esperar su turno para actualizar las estructuras globales, el tiempo de ejecución aumenta considerablemente. En cambio, la versión OpenMP-MultiLock utiliza locks de grano fino presenta mejores tiempos de ejecución, pero peores a las versiones RTM.

El servidor xeongold dispone de 32 cores y la posibilidad de utilizar hyperthreading. Al utilizar hyperthreading, se reparten los recursos entre dos threads y se obtienen peores resultados que al utilizar los recursos para un sólo thread. Esto es reflejado en las gráficas donde se puede ver el comportamiento de utilizar 64 threads en las diferentes versiones. Y en todas se obtienen peores tiempos de ejecución que utilizando 32 sin hyperthreading. Por lo que se seduce que no es recomendable el uso de hyperthreading en este tipo de algoritmos.

Las Figuras 12 y 13 muestran los speed-up utilizando 3 series temporales más grandes, descritas en la Figura 3, de un tamaño de 1.800.000 datos. En éstas, se comparan los speed-up de la versión RTM-MultiTransaccion y la versión Privado. Se puede observar que el uso

de series temporales de mayor tamaño mejora el comportamiento de las versiones RTM ya que existen menos interacciones entre los diferentes threads. Por ejemplo la relación entre la versión Privado y RTM-MultiTransaccion utilizando 32 threads con la serie temporal seysmology (180.000 datos) es de 3.3 veces mejor, en cambio utilizando la misma serie temporal con (1.800.000 datos) es de 1.4 veces mejor. Por lo que a medida que aumentamos el tamaño de la serie temporal se espera que los tiempos de ejecuciones sean similares, con la ventaja, en la versión RTM-MultiTransaccion, de ahorrar una cantidad de memoria considerable.

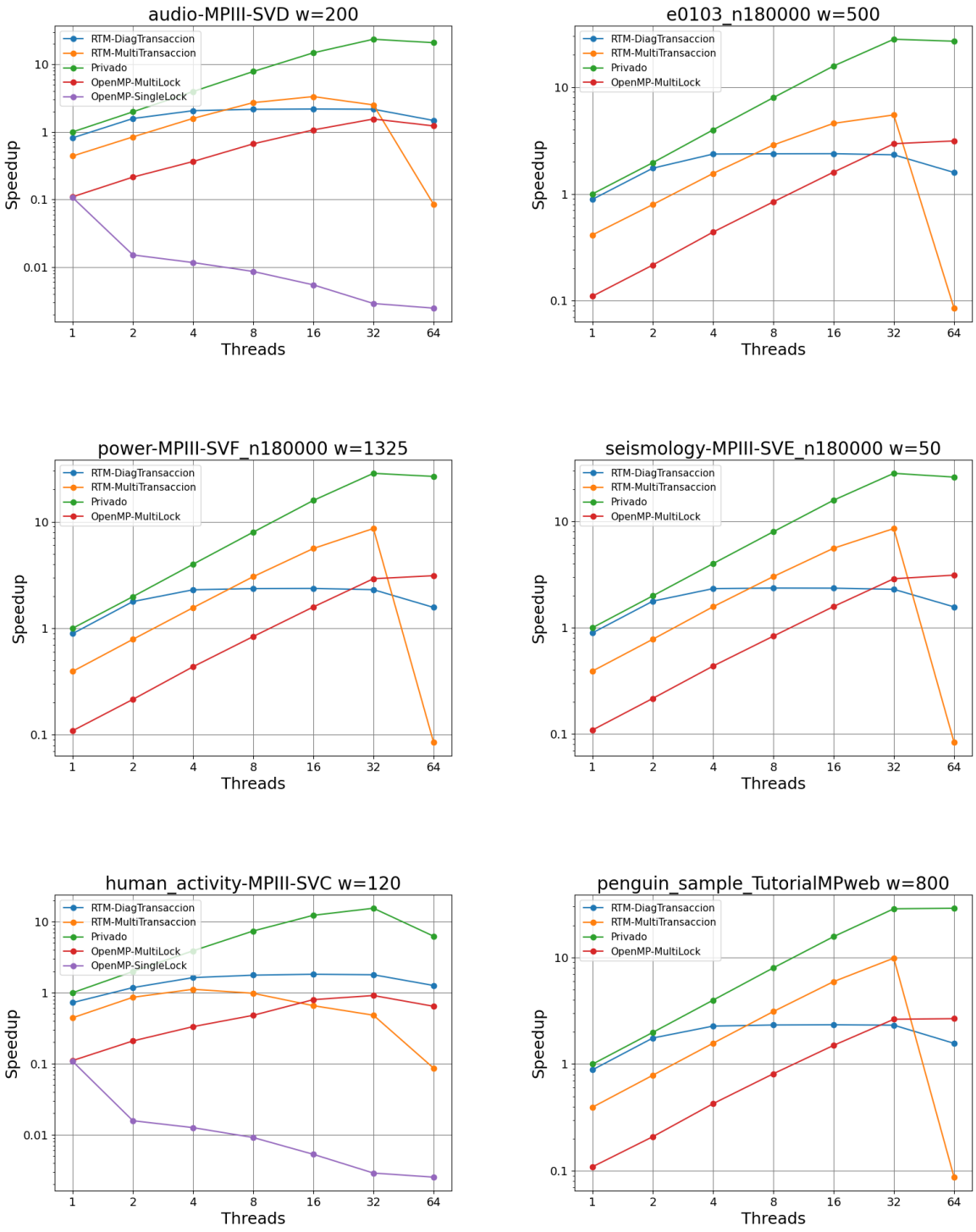

Figura 10: Resultados utilizando SCRIMP.

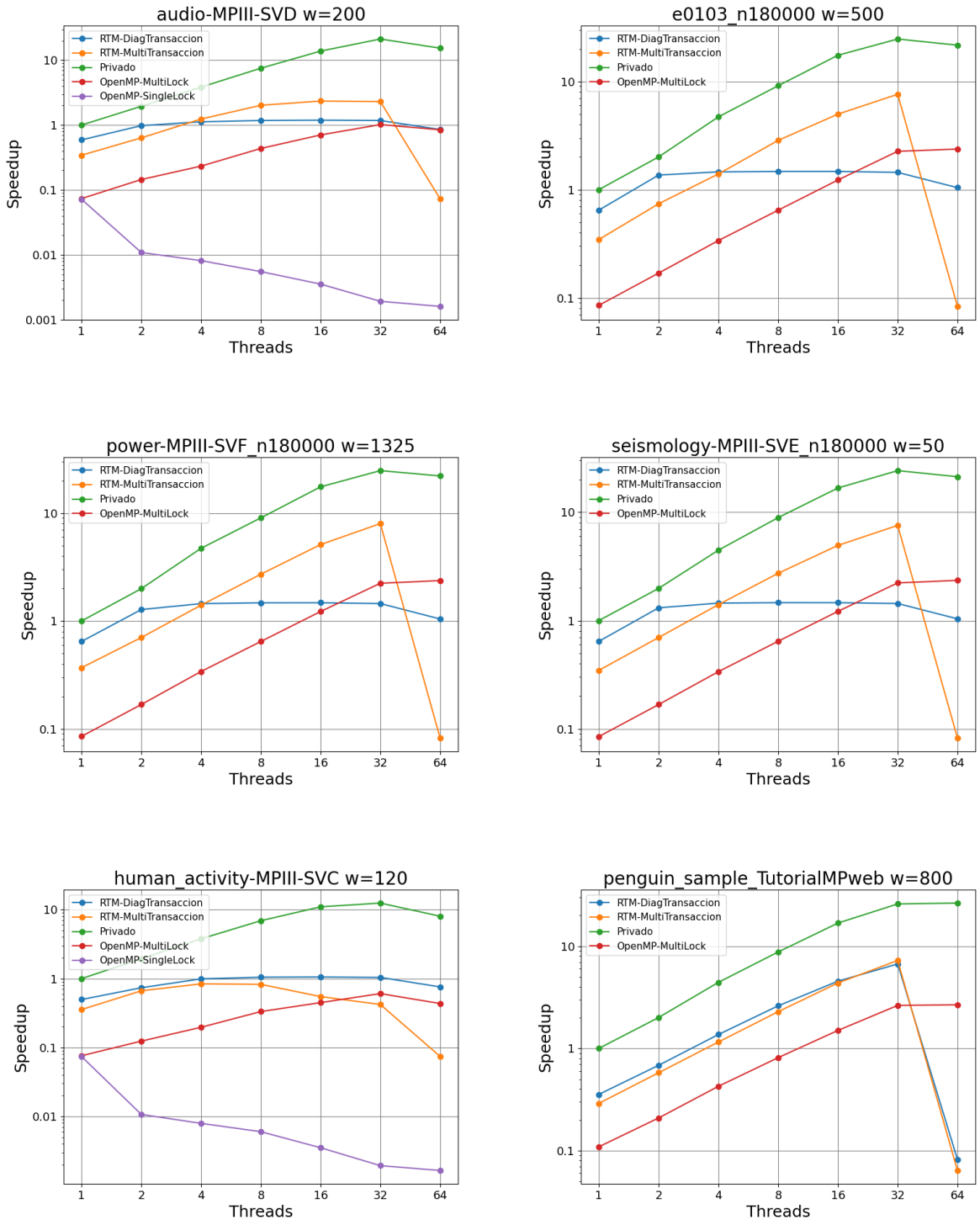

Figura 11: Resultados utilizando SCAMP.

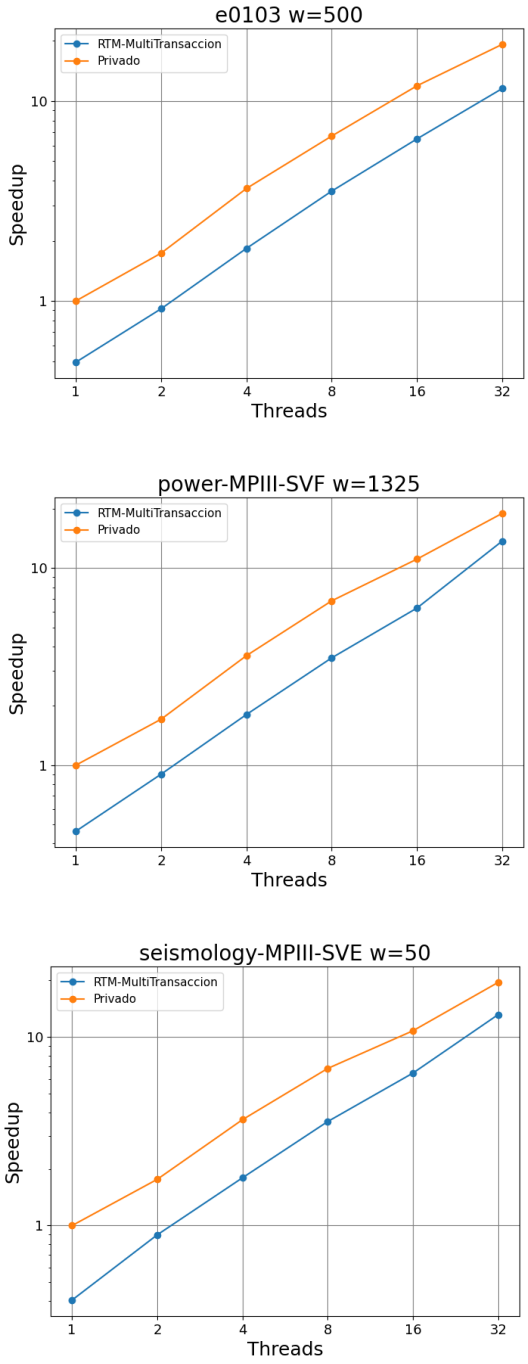

Figura 12: Resultados utilizando SCRIMP y las series temporales grandes.

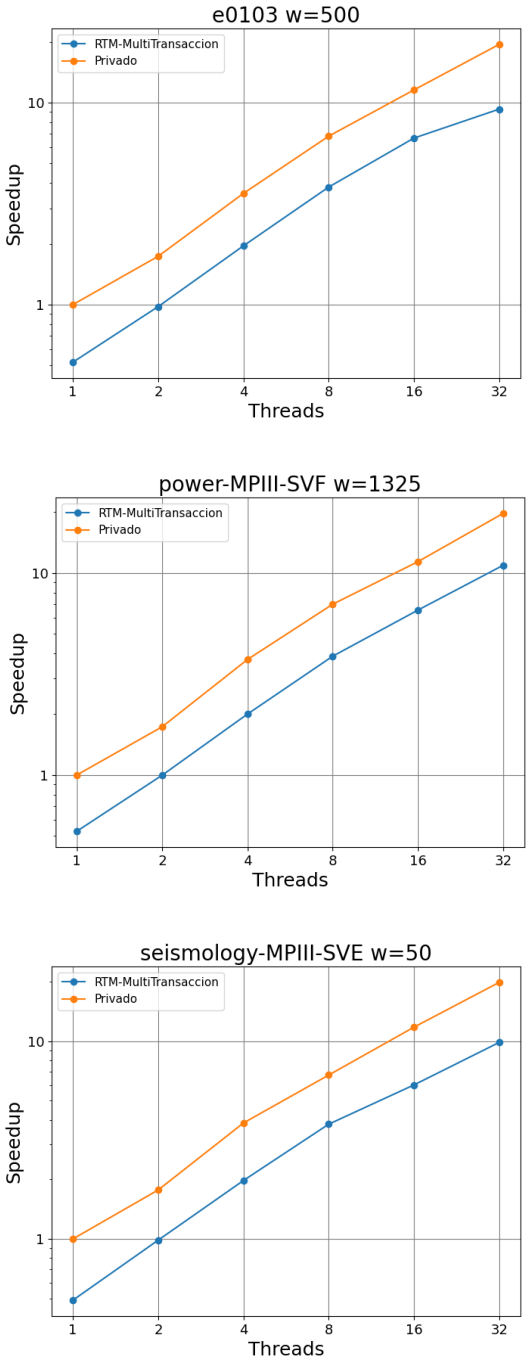

Figura 13: Resultados utilizando SCAMP y las series temporales grandes.

4.3.1. Consumo de memoria

Esta sección se presentan los diferentes consumos de memoria que utilizan las versiones alterando el número de threads utilizados y usando 3 series temporales de tamaños: 20.000, 180.000 y 1.800.000 . La Figura 14 utiliza una serie temporal de 20.000 datos. La Figura 15 utiliza una serie temporal de 180.000 datos. Y la Figura 16 utiliza una serie temporal de 1.800.000 datos.

Los valores de las Figuras 14, 15 y 16, son el consumo de memoria extra, es decir, a la hora de calcular el consumo de memoria se omite el consumo de memoria base que todos las versiones comparten. Éste consumo de memoria base se compone de una estructura de datos de tipo real para almacenar la serie temporal, unos vectores con datos de tipo real, que almacenan las variables estadísticas de tamaño profileLength y las estructuras globales profile y profile index, que almacenan datos de tipo real y enteros respectivamente, de tamaño profileLength. El consumo extra es determinado por las estructuras añadidas en cada versión en las funciones SCRIMP() y SCAMP(), que son utilizadas para calcular el profile y profile index global.

Destacar que para calcular el consumo de la versión RTM-DinamicTransaccion, se ha utilizado el tamaño máximo de transacción. Por ésta razón el consumo de memoria es igual al de la versión RTM-DiagTransaccion.

La versión Privado duplica las estructuras globales para cada thread, por lo que utilizando 32 threads, utilizaría 32 estructuras privadas además de las estructuras finales, por lo que cuanto más grande sea la serie temporal y más threads se quieran utilizar, peor será el problema de la memoria. Se observa en la Figura 16 que utilizando una serie temporal de 1.800.000 datos y 32 threads, la versión Privado utiliza 922 MB más que la versión RTM-MultiTransaccion. Pero si se utiliza una serie temporal de 500.000.000 datos y 32 threads, la versión Privado utilizaría 256 GB más que la versión RTM-MultiTransaccion.

Threads / Versiones	1	2	4	8	16	32	64
RTM-UniTransaccion	0B	0B	0B	0B	0B	0B	0B
RTM-MultiTransaccion	0B	0B	0B	0B	0B	0B	0B
RTM-DiagTransaccion	160KB	320KB	640KB	1.28MB	2.56MB	5.12MB	10.24MB
RTM-DinamicTransaccion	160KB	320KB	640KB	1.28MB	2.56MB	5.12MB	10.24MB
OpenMP-SingleLock	0B	0B	0B	0B	0B	0B	0B
OpenMP-MultiLock	80KB	80KB	80KB	80KB	80KB	80KB	80KB
Privado	320KB	640KB	1.28MB	2.56MB	5.12MB	10.24MB	20.48MB

Figura 14: Consumo de memoria extra utilizando una serie temporal de 20.000 datos.

Threads / Versiones	1	2	4	8	16	32	64
RTM-UniTransaccion	0B	0B	0B	0B	0B	0B	0B
RTM-MultiTransaccion	0B	0B	0B	0B	0B	0B	0B
RTM-DiagTransaccion	1.44MB	2.88MB	5.76MB	11.52MB	23.04MB	46.08MB	92.16MB
RTM-DinamicTransaccion	1.44MB	2.88MB	5.76MB	11.52MB	23.04MB	46.08MB	92.16MB
OpenMP-SingleLock	0B	0B	0B	0B	0B	0B	0B
OpenMP-MultiLock	720KB	720KB	720KB	720KB	720KB	720KB	720KB
Privado	2.88MB	5.76MB	11.52MB	23.04MB	46.08MB	92.16MB	184.32MB

Figura 15: Consumo de memoria extra utilizando una serie temporal de 180.000 datos.

Threads / Versiones	1	2	4	8	16	32	64
RTM-UniTransaccion	0B	0B	0B	0B	0B	0B	0B
RTM-MultiTransaccion	0B	0B	0B	0B	0B	0B	0B
RTM-DiagTransaccion	14.4MB	28.8MB	57.6MB	115.2MB	230.4MB	460.8MB	921.6MB
RTM-DinamicTransaccion	14.4MB	28.8MB	57.6MB	115.2MB	230.4MB	460.8MB	921.6MB
OpenMP-SingleLock	0B	0B	0B	0B	0B	0B	0B
OpenMP-MultiLock	7.2MB	7.2MB	7.2MB	7.2MB	7.2MB	7.2MB	7.2MB
Privado	28.8MB	57.6MB	115.2MB	230.4MB	460.8MB	921.6MB	1.8432GB

Figura 16: Consumo de memoria extra utilizando una serie temporal de 1.800.000 datos.

4.3.2. Discusión de los resultados

Una posible solución para calcular el Matrix Profile y Matrix Profile index sería la siguiente: en caso de querer obtener los resultados de una serie temporal pequeña, se podría utilizar la versión Privado, que es la más utilizada, tiene un buen tiempo de ejecución y al ser una serie temporal pequeña, el gasto de memoria sería razonable. A medida que se desee utilizar series temporales cada vez más grandes, las versiones RTM obtienen tiempos de ejecución cada vez más similares a la versión Privado, por lo que se podría utilizar, por ejemplo, la versión RTM-MultiTransaccion, con un tiempo de ejecución similar al utilizar series temporales grandes y con un gasto de memoria muy reducido. La tendencia de utilizar series temporales cada vez más grandes, genera un uso de memoria excesivo, en la versión Privado, por lo que el uso de las transacciones introducidas por RTM serían una buena solución, en la que se obtiene un tiempo de ejecución similar, cuanto más grande sea la serie temporal, y con la ventaja del reducido uso de memoria.

5
Conclusiones

El objetivo principal de este TFG era el de reducir el uso de recursos en implementaciones del Matrix Profile, utilizando Memoria Transaccional hardware, particularmente RTM, a la par que se mantiene un rendimiento adecuado.

Utilizando la versión RTM-MultiTransaccion se ha comprobado que se ha reducido el uso de recursos considerablemente con unos tiempos de ejecución similares a la versión privatizada, que es la versión más utilizada.

Además, se ha aportado una solución de privatización parcial con volcado de datos parciales al array global protegido por transacción (RTM-DinamicTransaccion) y una solución a una librería basadas en pragmas como OpenMP (OpenMP-SingleLock y OpenMP-MultiLock).

Los experimentos se han llevado a cabo utilizando series temporales reales y ejecutadas en un servidor Intel Xeongold, aportado por la UMA.

Se ha realizado un estudio de la sensibilidad del tamaño de la transacción, observando como afecta el aumento de las actualizaciones. Y se ha comprobado que es factible el uso de transacciones en el cálculo del Matrix Profile, aunque hay que valorar sus ventajas e inconvenientes.

Los algoritmos de búsquedas de patrones en series temporales como las implementaciones SCRIMP y SCAMP del Matrix Profile son importantes, sobre todo si se tiene en cuenta la tendencia a utilizar series temporales más grandes y esperar un tiempo de respuesta menor. Reducir el tiempo de ejecución permite aumentar la interactividad entre usuario y software, la cual es necesaria en diversos dominios como el de la medicina. Y tecnologías como RTM nos permiten tiempos de ejecución similares a la opción más utilizada con menores recursos.

Bibliografía

[1] Built-in functions for memory model aware atomic operations. https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html#g_t_005f_005fatomic-Builtins. Accedido sep. 2022.

[2] Extended asm - assembler instructions with c expression operands. https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html. Accedido sep. 2022.

[3] Intel Xeon Gold 6154. https://ark.intel.com/content/www/es/es/ark/products/120495/intel-xeon-gold-6154-processor-24-75m-cache-3-00-ghz.html. Accedido sep. 2022.

[4] Legacy __sync built-in functions for atomic memory access. https://gcc.gnu.org/onlinedocs/gcc/_005f_005fsync-Builtins.html. Accedido sep. 2022.

[5] OpenMP. https://gcc.gnu.org/onlinedocs/gccint/OpenMP.html. Accedido sep. 2022.

[6] The UCR Matrix Profile Page. https://www.cs.ucr.edu/~eamonn/MatrixProfile.html. Accedido sep. 2022.

[7] Transactional memory intrinsics. https://gcc.gnu.org/onlinedocs/gcc/x86-transactional-memory-intrinsics.html. Accedido sep. 2022.

[8] Burgess, M. *The GNU C Programming Tutorial*, 4.1 ed. 2002.

[9] Dau, H. A., and Keogh, E. Matrix Profile V: A Generic Technique to Incorporate Domain Knowledge into Motif Discovery. In *ACM SIGKDD Int'l. Conf. on Knowledge Discovery and Data Mining (KDD'17)* (2017), p. 125–134.

[10] Fernandez, I., Quislant, R., Gonzalez-Navarro, S., Gutierrez, E., and Plata, O. TraTSA: A Transprecision Framework for Efficient Time Series Analysis. *Journal of Computational Science 63* (2022), 15.

[11] Fernandez, I., Quislant, R., Gutiérrez, E., Plata, O., Giannoula, C., Alser, M., Gómez-Luna, J., and Mutlu, O. NATSA: A Near-Data Processing Accelerator for Time Series Analysis. In *Int'l. Conf. on Computer Design (ICCD'20)* (2020), pp. 120–129.

[12] Fernandez, I., Villegas, A., Gutierrez, E., and Plata, O. Accelerating time series motif discovery in the Intel Xeon Phi KNL processor. *The Journal of Supercomputing 75* (2019), 7053–7075.

[13] Harris, T., Larus, J., and Rajwar, R. *Transactional Memory, 2nd edition.* Morgan & Claypool Publishers, 2010.

[14] Herlihy, M., and Moss, J. E. B. Transactional memory: Architectural support for lock-free data structures. In *Int'l. Symp. on Computer Architecture (ISCA'93)* (1993), pp. 289–300.

[15] Intel. Intel 64 and ia-32 architectures optimization reference manual, 3 2022.

[16] Intel. Intel c++ compiler classic developer guide and reference, 4 2022.

[17] Prinz, U., and Prinz, P. *A Complete Guide to Programming in C++.* 2002.

[18] Quislant, R., Fernandez, I., Serralvo, E., Gutierrez, E., and Plata, O. Exploiting vector extensions to accelerate time series analysis. In *Euromicro Int'l. Conf. on Parallel, Distributed and Network-based Processing (PDP'22)* (2022), pp. 55–62.

[19] Quislant, R., Gutierrez, E., Zapata, E. L., and Plata, O. Insights into the Fallback Path of Best-Effort Hardware Transactional Memory Systems. In *Int'l. Conf. on Parallel Processing (Euro-Par'16)* (2016), pp. 251–263.

[20] Yeh, C.-C. M., Van Herle, H., and Keogh, E. Matrix Profile III: The Matrix Profile Allows Visualization of Salient Subsequences in Massive Time Series. In *Int'l. Conf. on Data Mining (ICDM'16)* (2016), pp. 579–588.

[21] Yeh, C.-C. M., Zhu, Y., Ulanova, L., Begum, N., Ding, Y., Dau, H. A., Silva, D. F., Mueen, A., and Keogh, E. Matrix Profile I: All Pairs Similarity Joins for Time Series: A Unifying View That Includes Motifs, Discords and Shapelets. In *Int'l. Conf. on Data Mining (ICDM'16)* (2016), pp. 1317–1322.

[22] Zhu, Y., Yeh, C.-C. M., Schall-Zimmerman, Z., Kamgar, K., and Keogh, E. J. Matrix Profile XI: SCRIMP++: Time Series Motif Discovery at Interactive Speeds. *Int'l. Conf. on Data Mining (ICDM'18)* (2018), 837–846.

[23] Zhu, Y., Zimmerman, Z., Senobari, N. S., Yeh, C.-C. M., Funning, G., Mueen, A., Brisk, P., and Keogh, E. Matrix Profile II: Exploiting a Novel Algorithm and GPUs to Break the One Hundred Million Barrier for Time Series Motifs and Joins. In *Int'l. Conf. on Data Mining (ICDM'16)* (2016), pp. 739–748.

[24] Zimmerman, Z., Kamgar, K., Senobari, N. S., Crites, B., Funning, G., Brisk, P., and Keogh, E. Matrix Profile XIV: Scaling Time Series Motif Discovery with GPUs to Break a Quintillion Pairwise Comparisons a Day and Beyond. In *ACM Symp. on Cloud Computing (SoCC'19)* (2019), p. 74–86.

Apéndice A
Manual de Experimentación

En esta sección se describe paso a paso los procedimientos necesarios para realizar las distintas pruebas, desde el acceso a los servidores hasta la ejecución de los experimentos.

A.1. Acceso al servidor

Para acceder al servidor remoto xeongold, utilizamos la Instrucción 19. Esta instrucción crea un túnel entre el puerto local especificado y el servidor xeongold a través del servidor de login y termina creando una conexión con el servidor. Para tener acceso hace falta usuario, contraseña y los permisos necesarios. Realizamos un túnel para facilitar el posterior traspaso de archivos desde el directorio local al directorio remoto del servidor utilizando otro shell. Se puede traspasar los archivos de una carpeta local al directorio remoto utilizando la Instrucción 20 y en caso contrario, traspasar los archivos del directorio remoto a una carpeta local utilizando la Instrucción 21. En ambas instrucciones se hace uso del túnel creado anteriormente especificando el puerto y la dirección localhost.

```
ssh -t -L puerto:xeongold.ac.uma.es:22 usuario@login.ac.uma.es ssh -t usuario@xeongold.ac
    .uma.es
```

Código 19: Creación de túnel y conexión al servidor xeongold.

```
scp -r -P puerto carpetaLocal/* usuario@127.0.0.1:/directorio/remoto
```

Código 20: Traspaso de archivos del directorio local al directorio remoto.

```
scp -r -P puerto usuario@127.0.0.1:/directorio/remoto/* carpetaLocal/
```

Código 21: Traspaso de archivos del directorio remoto al directorio local.

A.2. Compilación

Una vez se ha accedido al servidor, se puede compilar los programas utilizando los archivos makefile descritos en los Códigos 22 y 23.

El Código 22 es utilizado para compilar los archivos en los que no se utiliza RTM.

El Código 23 es utilizado para compilar los archivos en los que se utiliza RTM.

```
CC=g++
.PHONY: all clean

all: scamp scrimp

scamp: scamp.cpp
  $(CC) -O2 -Wall -o scamp scamp.cpp -lm -fopenmp -std=c++11


scrimp: scrimp.cpp
  $(CC) -O2 -Wall -o scrimp scrimp.cpp -lm -fopenmp -std=c++11


clean:
  rm -f *.o scrimp scamp
```

Código 22: Makefile utilizado para compilar versiones no RTM.

```
CC=g++
.PHONY: all clean

all: scamp scrimp

scamp: transaction.o scamp.o
  $(CC) -O2 scamp.o transaction.o -o scamp -fopenmp -lpthread -mrtm -lm

transaction.o: transaction.c transaction.h
  $(CC) -O2 transaction.c -mrtm -c

scamp.o: scamp.cpp transaction.h
  $(CC) -O2 -Wall scamp.cpp -lm -fopenmp -std=c++11 -mrtm -c


scrimp: transaction.o scrimp.o
  $(CC) -O2 scrimp.o transaction.o -o scrimp -fopenmp -lpthread -mrtm -lm

scrimp.o: scrimp.cpp transaction.h
  $(CC) -O2 -Wall scrimp.cpp -lm -fopenmp -std=c++11 -mrtm -c

clean:
  rm -f *.o scrimp scamp
```

Código 23: Makefile utilizado para compilar versiones RTM..

A.3. Screen

Para poder asegurar que la ejecución continúe en caso de problemas técnicos locales, lo cual cerraría la sesión ssh conectada con xeongold y pararía la ejecución, se utiliza la instrucción $screen$. La instrucción $screen$ permite crear una sesión en la cual se puede crear ventanas. Las ejecuciones lanzadas en las ventanas, no pararán aunque estas ventanas no estén visibles o terminemos la sesión ssh conectada al servidor. La Instrucción 24 crea una sesión con el nombre que deseemos. Una vez dentro de la sesión se puede realizar cualquier ejecución y para volver a la sesión ssh utilizamos el atajo $Ctrl + A + D$. De esta forma la ventana seguiría ejecutándose y se puede volver a ligarse a ésta de nuevo utilizando la Instrucción 25 y si es necesario ver el nombre de la ventana la Instrucción 26. Una vez finaliza la ejecución se puede cerrar la ventana utilizando la Instrucción 27.

```
screen -S nombreSesion -h 5000
```

Código 24: Intrucción para crear una sesión con una ventana.

```
screen -r nombreSesion
```

Código 25: Instrucción para ligarse a una ventana.

```
screen -ls
```

Código 26: Instrucción para ver las distintas ventanas creadas.

```
screen -XS nombreSesion quit
```

Código 27: Intrucción para cerrar una ventana.

A.4. Scripts de ejecución

Una vez dentro de una ventana creada con $screen$ se puede ejecutar los scripts que automatizarán la ejecución de los experimentos.

Para realizar el análisis del tamaño de transacción en las versiones RTM-MultiTransaccion y RTM-DinamicTransaccion, se ha utilizado el script 28 para ejecutar el programa utilizando las series temporales descritas en 3, distintos tamaños de transacción (desde 1 hasta 524288) y un thread. Se ejecuta 4 veces para evitar anomalías en los resultados y se elige la ejecución con menor tiempo de ejecución.

```
#!/bin/bash


benchs=("../timeseries/audio-MPIII-SVD.txt 200"
        "../timeseries/e0103_n180000.txt 500"
        "../timeseries/power-MPIII-SVF_n180000.txt 1325"
        "../timeseries/seismology-MPIII-SVE_n180000.txt 50"
        "../timeseries/human_activity-MPIII-SVC.txt 120"
        "../timeseries/penguin_sample_TutorialMPweb.txt 800")


transacciones="1 2 4 8 16 32 64 128 256 512 1024 2048 4096 8192 16384 32768 65536 131072
    262144 524288"

hilos="1"
for x in {1..4}; do
  for i in "${benchs[@]}"; do
    for t in $hilos; do
      for y in $transacciones; do
        echo "## $i $t $y"
        ./scrimp $i $t $y
        ./scamp $i $t $y
      done;
    done;
  done;
done;
```

Código 28: Script para ejecutar experimentos utilizados en el análisis de sensibilidad.

Una vez calculado el tamaño de transacción óptimo, se ejecuta el script 29 para todas las versiones. En este scripts a diferencia del anterior, variamos el número de threads (desde 1 hasta 64).

```
#!/bin/bash


benchs=("../timeseries/audio-MPIII-SVD.txt 200"
        "../timeseries/e0103_n180000.txt 500"
        "../timeseries/power-MPIII-SVF_n180000.txt 1325"
        "../timeseries/seismology-MPIII-SVE_n180000.txt 50"
        "../timeseries/human_activity-MPIII-SVC.txt 120"
        "../timeseries/penguin_sample_TutorialMPweb.txt 800")


hilos="1 2 4 8 16 32 64"

for x in {1..4}; do
  for i in "${benchs[@]}"; do
    for t in $hilos; do
      echo "## $i $t"
      ./scrimp $i $t
      ./scamp $i $t
    done;
  done;
done;
```

Código 29: Script para ejecutar experimentos.

Y para ejecutar los experimentos que utilizan las series temporales grandes, se utiliza el script 30.

```
#!/bin/bash


benchs=("../timeseries/power-MPIII-SVF.txt 1325"
        "../timeseries/e0103.txt 500"
        "../timeseries/seismology-MPIII-SVE.txt 50")


hilos="1 2 4 8 16 32"

for x in {1..4}; do
  for i in "${benchs[@]}"; do
    for t in $hilos; do
      echo "## $i $t"
      ./scrimp $i $t
      ./scamp $i $t
    done;
  done;
done;
```

Código 30: Script para ejecutar experimentos utilizando las series temporales grandes.

Apéndice B
Gráficas

Para obtener las gráficas se ha divido el trabajo en 3 archivos diferentes:

- **parse.py :** Archivo que describe varias clases que traducen los datos del fichero de estadísticas creado por SCRIMP.cpp o SCAMP.cpp y los almacena en diferentes arrays. Estos arrays serán utilizados para representar las gráficas.
- **Sensibilidad.py :** Archivo que ,utilizando el archivo parse.py, crea las diferentes gráficas utilizadas en la Sección 4.2.
- **Resultados.py :** Archivo que ,utilizando el archivo parse.py, crea las diferentes gráficas utilizadas en la Sección 4.3.

B.0.1. Parse.py

En este archivo de definen las siguientes clases:

- **StatsFile :** Utilizando las librerías $fnmatch$ y re para expresiones regulares, creamos las diferentes expresiones que serán utilizadas para la lectura de los archivos de estadísticas. Realizando la lectura se comprueba línea a línea si coincide con la expresión regular y en caso de éxito se actualiza la variable correspondiente.
- **StatsAvgThreads :** Lee los datos del tiempo y número de threads de todos los ficheros expecificados por argumento utilizando la clase StatsFile y los almacena en dos arrays.
- **StatsAvgTransaction :** Lee los datos del tiempo y número de actualizaciones de todos los ficheros especificados por argumento utilizando la clase StatsFile y los almacena en dos arrays.

B.0.2. Sensibilidad.py

Este archivo se encarga de la ejecución del programa, contiene la función main donde utilizando el archivo parse.py crea las gráficas, mostradas en la Sección 4.2, a partir de los resultados previamente calculados de los algoritmos.

B.0.3. Resultados.py

Este archivo se encarga de la ejecución del programa, contiene la función main donde utilizando el archivo parse.py crea las gráficas, mostradas en la Sección 4.3, a partir de los resultados previamente calculados de los algoritmos.

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga

E.T.S. DE INGENIERÍA INFORMÁTICA