Exploring Multiprocessor Approaches to Time Series Analysis

Ricardo Quislant[a], Eladio Gutierrez[a], Oscar Plata[a]

[a]*Dept. Computer Architecture, University of Málaga, Málaga, Spain*

Abstract

Time series analysis is a key technique for extracting and predicting events in domains as diverse as epidemiology, genomics, neuroscience, environmental sciences, economics, etc. *Matrix Profile*, a state-of-the-art algorithm to perform time series analysis, finds out the most similar and dissimilar subsequences in a time series in deterministic time and it is exact. *Matrix Profile* has low arithmetic intensity and it operates on large amounts of time series data, which can be an issue in terms of memory requirements. On the other hand, Hardware Transactional Memory (HTM) is an alternative optimistic synchronization method that executes transactions speculatively in parallel while keeping track of memory accesses to detect and resolve conflicts.

In this work, we evaluate one of the best implementations of Matrix Profile exploring several multiprocessor variants to it and proposing new implementations taking into account a variety of synchronization methods (HTM, locks, barriers) and algorithm organizations. We analyze these matrix profile variants with real datasets (short and large) in terms of speedup and memory requirements, the latter being a major issue when dealing with very large time series. The experimental evaluation shows that our proposals can get up to 80× speedups for 128 threads, while keeping memory requirements low and even not dependent on the number of threads.

Keywords: Time Series Analysis, Matrix Profile, Hardware Transactional Memory, Shared-Memory Parallelism, Tiling, Intel Restricted Transactional Memory (RTM)

1. Introduction

A time series is a chronologically ordered set of samples of a real-valued variable that can have millions of observations. Time series analysis seeks extracting models in a large variety of domains [1] such as epidemiology, DNA analysis, economics, geophysics, speech recognition, etc. Particularly, *motif* [2] (similarity) and *discord* [3] (anomaly) discovery has become one of the most frequently used primitives in time series data mining [4, 5, 6, 7, 8, 9]. It poses the problem of solving the all-pairs-similarity-search (also known as similarity join). Specifically, given a time series broken down into subsequences, retrieve the most similar subsequences (motifs) and the most different ones (discords).

One of the state-of-the-art methods for motif and discord discovery is *Matrix Profile* [10]. It solves the similarity join problem and allows time-manageable computation of very large time series. In this work, we focus on this technique, which features the possibility of detecting similarities, anomalies, and predicting outcomes. It provides full joins without the need for specifying a similarity threshold, which is a very challenging task in this domain. The matrix profile is another time series representing the minimum distance subsequence for each subsequence in the time series (motifs). Maximum distance values of the profile highlight the most dissimilar subsequences (discords).

On the other hand, Transactional Memory (TM) [11, 12, 13] is an alternative to locking techniques aimed at simplifying parallel programming. A transaction is a section of code that is guaranteed to execute atomically and isolated. The TM system executes transactions speculatively in parallel while keeping track of memory accesses to detect and resolve conflicts. Thus, TM is considered to provide optimistic concurrency control as opposed to the pessimistic lock-based way of dealing with critical sections, which always serializes the execution. Many TM proposals arose in the last two decades, including both software (STM) [14, 15] and hardware designs (HTM) [16, 17, 18]. In the last years, major manufacturers of commercial processors have added HTM extensions to their architectures [19, 20, 21]. Such extensions are called *best-effort* HTMs because they do not offer finalization guarantees for transactions.

In this work, we evaluate one of the best implementations of matrix profile (SCAMP [22]) exploring several multiprocessor variants to it and proposing new implementations taking into account a variety of synchronization methods (HTM, locks, barriers) and algorithm organizations. We analyze these matrix profile variants with real datasets (short and large) in terms of speedup and memory requirements, since the latter can be a major issue when dealing with very large time series [23, 24, 25]. Even though Matrix Profile claims to attain a reduced memory footprint [26], it can rise rapidly with the number of threads in multiprocessor implementations as an effect of privatization.

The contributions of this work are the following:

- We explore a great variety of multiprocessor approaches to Matrix Profile from fine grain locking to HTM-based synchronization, and based on tiling organization as well.
- We propose two new implementations based on HTM. With one of them (the one using tiling) yielding the best

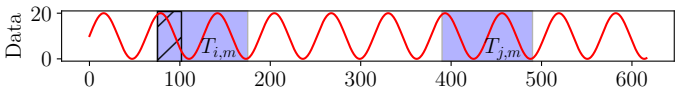

Figure 1: Example of two subsequences, $T_{i,m}$ and $T_{j,m}$, of a given time series. When computing the matrix profile P, subsequences starting in the exclusion zone of $T_{i,m}$ are ignored for their high similarity.

performance and low memory requirements, and the other one providing the lowest memory requirements.

- We propose a new tiling approach using barriers that shows both remarkable speedup when using large time series and the lowest memory requirements independently from either the number of threads used or the tile size.
- We come up with a heuristic to choose the proper tile size and thread count depending on the subsequence length and the time series length, in order to outperform the baseline implementation of the Matrix Profile.

Our proposals show speedups of up to 70× and 80× over sequential with 128 threads, and memory requirements not depending on the number of threads spawned by the application.

The remainder of the work is organized as follows. Section 2 introduces Matrix Profile time series analysis and HTM. Section 3 describes the variety of approaches to Matrix Profile we use and our different implementation proposals. Section 4 discusses the experimental evaluation outlining the methodology we follow, a transactional sensitivity analysis, and the results we obtained both in terms of speedup and memory requirements. Section 5 gives a related work overview and, finally, Section 6 concludes the work.

2. Background

2.1. Time Series Analysis. The Matrix Profile

A *time series* T is a sequence of n data points t_i, $1 \leq i \leq n$, collected over time. Let be $T_{i,m}$ a subsequence of T, where i is the index of its first data point, T_i, and m is the number of data points in the subset, with $1 \leq i$, and $m \leq n$. In the literature, $T_{i,m}$ is also called a *window* of length m. Fig. 1 shows an example of a time series with two subsequences highlighted, $T_{i,m}$ and $T_{j,m}$.

One way to measure the similarity between two time series subsequences is to use the *z-normalized Euclidean distance*, $d_{i,j}$, as done in SCRIMP [27], which is calculated as follows:

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

where $Q_{i,j}$ is the dot product of subsequences $T_{i,m}$ and $T_{j,m}$. μ_x and σ_x are the mean and the standard deviation of the data points in $T_{x,m}$, respectively.

We can use this distance measure to find the most similar subsequences out of all subsequences of a time series T. There are three steps to this procedure: (1) building a symmetric $(n - m + 1) \times (n - m + 1)$ matrix D, called *distance matrix*, with a window size m and a time series of length n. Each D cell, $d_{i,j}$,

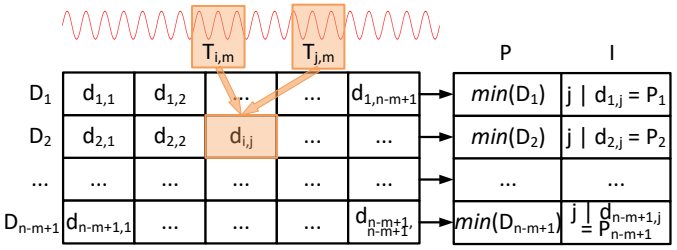

Figure 2: Computation of the matrix profile P and the profile index I from the distance matrix D. P_i is the minimum distance of each row (or column) D_i. I_i is the index of the subsequence providing such minimum.

stores the distance between two subsequences, $T_{i,m}$ and $T_{j,m}$. Thus, each row (or column) of D stores the distances between a subsequence of T and every subsequence of T; (2) finding the two subsequences whose distance is minimum (i.e., most similar sequences) in each row (or column) of D. This can be computed by building the *matrix profile*, P, which is a vector of size $n - m + 1$. Each cell P_i in P stores the minimum value found in the i^{th} row (or column) of D; (3) finding the indices of the most similar subsequences of the matrix profile. This requires building another vector I –called *matrix profile index*– of the same size as that of P, where $I_i = j$ if $d_{i,j} = P_i$. In this way, P contains the minimum distances between subsequences of T, while I is the vector of "pointers" to the location of these subsequences. Fig. 2 depicts an example of the distance matrix (D), the matrix profile (P), and the matrix profile index (I) for the time series in Fig. 1.

Matrix Profile algorithms are designed to store only the matrix profile and the matrix profile index arrays, computing the minimum distances $d_{i,j}$ on the fly. Note that the neighboring subsequences of $T_{i,m}$ are highly similar to it (i.e., $d_{i,i+1} \approx 0$) due to the overlaps. Thus, we can exclude these subsequences from computing D to find similar subsequences other than the neighboring ones. This is done by defining an exclusion zone (diagonally stripped zone in Fig. 1) for each subsequence. In general, the exclusion zone for a subsequence $T_{i,m}$ of length m is $\frac{m}{4}$ [27].

SCAMP [22] follows a similar computation scheme than SCRIMP, but instead replaces the sliding dot product with a mean-centered sum of products with the goal of reducing the number of operations required and the floating-point rounding errors. The following equations can be precomputed in $O(n-m+1)$ time, having the length of the profile vector defined as $n - m + 1 = l$:

$$df_i = \frac{T_{i+m-1} - T_{i-1}}{2}, \quad 0 < i < l \quad (2)$$

$$dg_i = T_{i+m-1} - \mu_i + T_{i-1} - \mu_{i-1}, \quad 0 < i < l \quad (3)$$

$$ssq_i = \begin{cases} \sum_{k=0}^{m-1}(T_k - \mu_0)^2, & i = 0 \\ ssq_{i-1} + (T_{i+m-1} - \mu_i + T_{i-1} - \mu_{i-1}) & \\ (T_{i+m-1} - T_{i-1}), & 0 < i < l \end{cases} \quad (4)$$

$$\sigma_i = \sqrt{ssq_i}, \quad 0 \leq i < l \quad (5)$$

Eqs. 2 and 3 are terms used in the covariance update of Eq. 6, and the standard deviation (L2-norm of subsequence $T_{i,m} - \mu_i$)

calculated in Eqs. 4 and 5 is used for the Pearson correlation coefficient depicted by Eq. 7. Note the exclusion zone in the limits of Eq. 6 given by $\frac{m}{4}$.

$$\sigma_{i,j} = \begin{cases} \sum_{k=0}^{m-1}(T_k - \mu_0)(T_{k+j} - \mu_j), & i = 0, \frac{m}{4} < j < l \\ \sigma_{i-1,j-1} + df_i dg_j + df_j dg_i, & i > 0, \frac{m+4}{4} < j < l \end{cases} \quad (6)$$

$$P_{i,j} = \frac{\sigma_{i,j}}{\sigma_i \sigma_j} \quad (7)$$

$$D_{i,j} = \sqrt{2m(1 - P_{i,j})} \quad (8)$$

The matrix profile can be derived incrementally for each diagonal of the distance matrix, Eq. 6, from the calculation of the covariance of two subsequences of the first row (first piece in Eq. 6). The Pearson correlation coefficient in Eq. 7 can be computed in fewer operations and it is more robust than the Euclidean Distance used by SCRIMP. Eq. 8 calculates the distance from the Pearson coefficient in $O(1)$.

2.2. Hardware Transactional Memory

Commercial HTM extensions [20, 21] implement a TM system based on caches and the coherence protocol, such as the seminal approach of Herlihy and Moss [11]. Core's private L1 cache is used to keep transactional updates, while L2 and L3 caches are used to keep track of transactionally read locations. Transactional read/write bits are provided with each cache block so that the coherence protocol can check for conflicts among transactions. On a transactional read request, the coherence protocol checks these transactional bits. If the write bit was set by another transaction, the protocol sends an abort command either to the requester (requester-loses conflict resolution policy) or to the requested thread (requester-wins conflict resolution policy). Requests from non-transactional code to transactional blocks favor non-transactional code enforcing the strong-isolation property [28].

On the Instruction Set Architecture (ISA) level, HTM extensions provide a concise set of instructions to manage transactions: *xBegin(ret)* marks the beginning of a transaction and may receive a label, *ret*, that points to the line of code we want the thread to resume on abort. From *xBegin()* onwards, the HTM system starts the bookkeeping of memory accesses in the cache hierarchy; *xCommit()* marks the end of a transaction, resets transactional bits and releases private updates. An abort instruction explicitly aborts a transaction from within. HTMs with support for escape actions [29, 21] add the pair *escapeBegin()* and *escapeEnd()* to temporarily disable TM tracking in the code enclosed between them.

This HTM implementation results in a *best-effort* system where a transaction aborts whenever it runs out of bookkeeping hardware, risking live-lock because of never-ending abort and retry. To ensure forward progress the user must implement a fallback mechanism which often consists of the transaction body protected by a global lock. Fig. 3 shows a simplified API for transactions that implements a fallback mechanism. The begin/commit instructions are replaced with the TX_START/TX_STOP wrappers which check a thread-local retry counter (line 2) to choose whether to execute the fallback or the transaction. First, the counter is incremented, line 4, and the transaction begins, line 8. If the transaction aborts, the execution is resumed in line 4 and the retry counter is incremented again. Once the number of retries is greater than MAX_RETRIES, line 5, the global fallback lock is taken and the execution is serialized, line 6. The transaction must subscribe to the global lock by reading it, so that it aborts when other thread runs the fallback code. Other optimizations [30] can be approached. At the end of the critical section the counter is checked again to see whether it is the transaction or the fallback executing, line 12. In the first case the transaction is committed, line 13. Otherwise, the fallback lock is released, line 15. Finally, the retry counter is reset in line 17.

```
1:  global fallbackLock ← 0                          ▷ Global lock for fallback execution
2:  thread local retries ← 0                         ▷ Retry counter for transactions
3:  procedure TX_START(retries)
4:      retries ← retries + 1
5:      if (retries > MAX_RETRIES) then
6:          setlock(fallbackLock)                    ▷ Fallback begin
7:      else
8:          xBegin(4)                                ▷ Go to line 4 on abort
9:      end if
10: end procedure
11: procedure TX_STOP(retries)
12:     if (retries ≤ MAX_RETRIES) then
13:         xCommit()                                ▷ Transaction end
14:     else
15:         unsetlock(fallbackLock)                  ▷ Fallback end
16:     end if
17:     retries ← 0                                  ▷ Reset retry counter
18: end procedure
```

Figure 3: Simplified API implementation for hardware transactions

3. Multiprocessor Matrix Profile Approaches

3.1. Baseline approach

Fig. 4 shows the SCAMP embarrassingly parallel algorithm pseudocode. Consider all variables as local to the procedure they firstly appear on. Global variables are tagged as such in the pseudocode. First, the length of the profile and profile index arrays is defined in terms of the length, n, of the time series, T, by subtracting to it the subsequence length, m. Subsequently, the global profile and profile index arrays are defined, as well as the profiles which will hold the statistics needed to calculate the covariances and correlations. It is worth noting that SCAMP uses privatization arrays per thread to get rid of the need for synchronization on the access to global arrays. Interestingly enough, those privatization arrays are defined as global as well, see line 5, as the final reduction will need each thread to access the private arrays of other threads. Consequently, each thread works with its own private profile and profile index, which is a chunk of the global private profile and profile index arrays, accessed with an offset, off in line 8, calculated from the thread's ID (from 1 to *thCount* in this case). Each thread executes the SCAMP procedure in line 7.

```
1:  global T, n, m, thCount;                                                  ▷ Application parameters
2:  global l ← n − m + 1;                                                     ▷ Profile length is T length, n, minus windows length, m, plus 1
3:  global profile[l], profileIndex[l];                                       ▷ Global double profile and long integer profile index arrays
4:  global μ[l], σ[l], df[l], dg[l];                                          ▷ Global double statistic arrays
5:  global privProfile[l∗thCount], privProfileIndex[l∗thCount];               ▷ Per-thread privatization arrays
6:  μ, σ, df, dg ← computeStatistics(T, m);                                   ▷ Precomputation of statistics
7:  procedure SCAMP                                                           ▷ thCount threads are created (e.g. #pragma omp parallel) that execute this procedure
8:      off ← (gettid() − 1) ∗ l;                                             ▷ Offset to access privatization arrays depending on thread ID (tid)
9:      privProfile[off+1:off+l] ← −∞;                                        ▷ Private profile initialization
10:     for diag ∈ [exclusionZone+1:l] do                                     ▷ Each thread takes diagonals on demand: e.g. #pragma omp for (dynamic)
11:         cov ← 0;                                                          ▷ Covariance of the first element in the diagonal
12:         for k ← 1 to m do                                                 ▷ Covariance calculated as a mean-centered sum of products
13:             cov ← cov + (T[diag+k] − μ[diag]) ∗ (T[k] − μ[1]);
14:         end for
15:         corr ← cov ∗ σ[1] ∗ σ[diag];                                      ▷ Correlation computation of the first element of the diagonal
16:         if corr > privProfile[off+1] then                                 ▷ Update of private matrix profile and index arrays
17:             privProfile[off+1] ← corr;
18:             privProfileIndex[off+1] ← diag;
19:         end if
20:         if corr > privProfile[off+diag] then                              ▷ Distance matrix is symmetric
21:             privProfile[off+diag] ← corr;
22:             privProfileIndex[off+diag] ← 1;
23:         end if
24:         i ← 2;
25:         for j ← diag + 1 to l do                                          ▷ Rest of the diagonal
26:             cov ← cov + df[i−1] ∗ dg[j−1] + df[j−1] ∗ dg[i−1];            ▷ Covariance is now incrementally calculated
27:             corr ← cov ∗ μ[i] ∗ μ[j];
28:             if corr > privProfile[off+i] then                             ▷ Update of private matrix profile and index arrays
29:                 privProfile[off+i] ← corr;
30:                 privProfileIndex[off+i] ← j;
31:             end if
32:             if corr > privProfile[off+j] then                             ▷ Distance matrix is symmetric
33:                 privProfile[off+j] ← corr;
34:                 privProfileIndex[off+j] ← i;
35:             end if
36:             i ← i + 1;
37:         end for
38:     end for
39:     barrier();                                                            ▷ Wait for all threads to finish before the final reduction
40:     for i ← 1 to l do                                                     ▷ Each thread is assigned a chunk of the global profile to reduce to: e.g. #pragma omp for (static)
41:         maxCorr ← −∞;
42:         for j ← 1 to thCount do                                           ▷ Search for the maximum correlation in all the privatization arrays
43:             off ← (j − 1) ∗ l;
44:             if privProfile[off+i] > maxCorr then
45:                 maxCorr ← privProfile[off+i];
46:                 maxIndex ← privProfileIndex[off+i];
47:             end if
48:         end for
49:         profile[i] ← maxCorr;                                             ▷ Set reduction results in global profile arrays
50:         profileIndex[i] ← maxIndex;
51:     end for
52: end procedure
```

Figure 4: SCAMP baseline embarrassingly parallel algorithm pseudocode.

The SCAMP procedure has a main outer loop, see line 10, where different threads are assigned iterations (diagonals of the distance matrix) on demand. The first inner loop, in line 12, calculates the covariance of the first element in the diagonal as a mean-centered inner product of the subsequences involved. Next, the correlation is computed and the private profile updated whenever the correlation is greater than the previously calculated for the subsequences, which is the case for the first element.

The second inner loop, in line 25, is in charge of computing the rest of the diagonal. In this loop, the covariance is computed incrementally from the value calculated earlier, which saves a huge amount of floating point operations. The private profile update comes next, which now may or may not store the correlation, depending on the time series values and the amount of subsequences bearing a strong resemblance.

The final part of the algorithm is a reduction from the private profile and profile index arrays to the global ones. As aforementioned, each thread is assigned a chunk of the global profile arrays and reduces the private values calculated by all threads to that global profile chunk. Before proceeding with this final reduction, all threads must have finished with their private profile

computations, thus the barrier in line 39.

3.2. Fine-grain locking and HTM implementations

Although the baseline algorithm is embarrassingly parallel, we wanted to explore two approaches that use thread synchronization other than barriers, to get rid of the privatization arrays (less memory needed) so that there is no need for the final reduction, i.e. they work with the global arrays directly. Fine-grain locking (FGL) and hardware transactional memory (HTM) were explored for that purpose.

The FGL approach consists in defining a global array of locks with a lock per profile array element: **global** $fgLock[l]$. Then, each update of the private profile is changed so that it updates the global profile as follows:

```
...
setlock(fgLock[i]);
if corr > profile[i] then   ▷ Update of global matrix profile and index arrays
   profile[i] ← corr;
   profileIndex[i] ← j;
end if
unsetlock(fgLock[i]);
...
```

Each if is protected with its fine grain lock. The final reduction is consequently erased.

The HTM approach does not need an array of locks. Unlike locks, transactions are critical sections without a variable associated to it, so we only need to start a transaction before a profile update and stop it afterwards using the API in Fig. 3. However, transactions may have a limited footprint since HTM systems are best-effort. That means that the transactional hardware may only keep track of a certain amount of reads and writes before aborting a transaction because of capacity. Also, a transaction has a cost to it, derived from the fact that there are implicit memory barriers on opening and closing it [31], and the processor pipeline has to be reset on each abort. Then, it can be useful to have a TM implementation where the size of the transaction can be parameterized. In doing so, we propose a variant of the baseline algorithm with neither private profile arrays nor final reduction phase, and where the global profile updates are protected with transactions of configurable size based on the XACT_SIZE parameter:

```
...
TX_START(retries);
if corr > profile[i] then                ▷ Global profile and index update
   profile[i] ← corr;
   profileIndex[i] ← j;
end if
if corr > profile[j] then                ▷ Global profile and index update
   profile[j] ← corr;
   profileIndex[j] ← i;
end if
xactCount ← xactCount + 1;
if xactCount = XACT_SIZE then
   TX_STOP(retries);
   xactCount ← 0;
   TX_START(retries);
end if
...
```

Each i and j elements of the global matrix profile and index arrays are updated inside a transaction, and the thread local variable *xactCount* is incremented. The transaction commits only if the number of updates (both i and j) is equal to the XACT_SIZE parameter. We carry out a transaction size sensitivity analysis in Section 4.2 in order to find the largest transaction size without saturating the capacity of the transactional hardware.

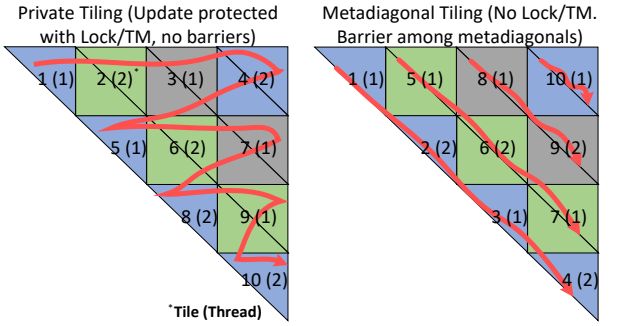

Figure 5: Private tiling versus metadiagonal tiling. Example of tile-to-thread assignment and synchronization with two threads.

3.3. Private tiling approaches with locks and HTM

For our private tiling approaches we follow a similar tiling scheme as that used in [22] for their GPU-STOMP and GPU-SCAMP algorithms. Fig. 6 outlines the pseudocode of our lock-based version. The distance matrix in Fig. 2 is divided into tiles of a given size L (given in elements of the tile's edge), which is passed as application parameter (Fig. 6, line 1). Then, threads define private partial matrix profile and index arrays the size of a tile (for the i and j parts of the tile - lines 9 to 11), and threads are assigned tiles on demand (line 12). After computing its tile, each thread updates the global matrix profile and index arrays within a critical section protected with a global lock or transactions. Such an update, other than our approaches are implemented for CPUs instead of GPUs, is the main difference with the algorithms in [22], which update the global array with atomic instructions (such as compare-and-swap). They make it possible because they pack together the matrix profile value and the index into a 64bit structure holding a float correlation value and a 32bit integer index. Conversely, our approaches handle double profile values and unsigned long integer indexes providing more precision and the capability of dealing with series longer than 4G elements.

On the left of Fig. 5 we can see the tiles and its numbering and how they are assigned to threads. Each thread computes a whole tile comprising upper and lower triangles (Fig. 6, lines 13 to 16). However, the tiles in the main metadiagonal [1] comprises upper triangles only (line 14) and may introduce load imbalance. Furthermore, the last tile in a metarow might be smaller depending on the profile length, l, not being multiple of the tile size, L. Hopefully, that should not be a problem as tiles in a metarow are assigned to threads on demand, and there is no barrier before starting off with a new metarow of tiles.

The size of the tile, L, will have an impact on the performance and precision of the algorithm. Computing a tile's diagonal implies computing the covariance and correlation of the

[1] A metadiagonal or a metarow are a diagonal or a row made of tiles

first element in the diagonal as the mean-centered sum of products in lines 12-15 of Fig. 4, and, subsequently, computing the covariance and correlation of the rest of the elements in the diagonal incrementally from the first element (loop in line 25 of Fig. 4). A small tile size implies many more operations due to the first element computation of tile's diagonals. Conversely, as the incremental part of the diagonal computation is shortened, the error carried by floating point operations is reduced and the algorithm is more precise. We discuss a tile size exploration in Section 4.3.1.

Finally, as aforementioned, we explore two flavors of our private tiling algorithm. The first one entails a global lock to protect the update from the private tile arrays to the global arrays (Fig. 6, lines 17 to 30). It is worth noting that the values stored into the private tile arrays have an offset with respect to their actual place in the global arrays. The corresponding offset is calculated by the functions getIoffset and getJoffset depending on the tile number. In case of the HTM version, that snippet of code is protected by a transaction. The HTM version also has the size of the transaction as an application parameter so that we can set the proper transaction size that does not overflow the transactional hardware. If such a size is lower than the tile size several transactions will be needed to perform the update from the private tile arrays to the global arrays.

3.4. Metadiagonal tiling implementation

Our metadiagonal tiling algorithm is a proposal which implies tiling but does not need private arrays to perform the tile computation. Conversely, all updates are carried out on the global arrays and there is no need for lock or HTM synchronization. It is made possible by how we assign tiles to threads. Fig. 5 shows such a tile assignment compared with the private tiling approaches discussed in the previous section. Each thread picks a tile on demand from the tiles in a metadiagonal. A metadiagonal is a diagonal made of tiles. When there are no more tiles to compute in the metadiagonal, threads wait at a barrier for other threads to finish their work before starting off with another metadiagonal. Consequently, we need barriers to synchronize threads but, as tiles in a metadiagonal comprise disjoint elements of the profile both on the *i* and *j* dimensions (see the saw-tooth shape of a metadiagonal in Fig. 5), we do not need either locks or transactions to protect the update.

Our metadiagonal tiling algorithm is similar to that in Fig. 6 but without the private tile arrays. The lock/HTM protected snippet of code in lines 17 to 30 is no longer needed either, and the computation of the lower and upper triangles in a tile (lines 13 to 16) is changed to work with the global arrays directly. Finally, the for loop in line 12 must ensure that threads are assigned tiles in a metadiagonal on demand, and that a barrier is placed before starting off a new metadiagonal (e.g. OpenMP for nowait clause must be omitted).

While the private tiling approaches of the previous section could introduce low risk for load imbalance due to the different computational load of the different tiles (main metadiagonal and right-end metacolumn tiles), our metadiagonal approach could make such a risk more noticeable because of the barrier

```
 1: global T, n, m, thCount, L;                          ▷ Application parameters
 2: global l ← n − m + 1;                                 ▷ Profile length
 3: global gLock;                                         ▷ Global lock
 4: global profile[l], profileIndex[l];                   ▷ Profile and index arrays
 5: global μ[l], σ[l], df[l], dg[l];                      ▷ Statistic arrays
 6: μ, σ, df, dg ← computeStatistics(T, m);
 7: procedure SCAMP_TILES_LOCK          ▷ thCount threads are created (e.g. #pragma omp parallel) that execute this procedure
 8:                 ▷ Private tile initialization
 9:   privProfileI[1:L] ← −∞;
10:   privProfileJ[1:L] ← −∞;
11:   privProfileIndexI[L], privProfileIndexJ[L];
12:   for tile ∈ [1:⌈l/L⌉ ∗ (⌈l/L⌉ + 1)/2 ] do   ▷ Threads take tiles on demand: e.g. omp for (dynamic, nowait)
13:     computeUpperTriangle(tile, privProfileI, privProfileIndexI, privProfileJ, privProfileIndexJ);
14:     if ¬ isMainDiagonalTile(tile) then
15:       computeLowerTriangle(tile, privProfileI, privProfileIndexI, privProfileJ, privProfileIndexJ);
16:     end if
17:     setlock(gLock);                                ▷ Global profile and index update
18:     for i ← 1 to L do
19:       if privProfileI[i] > profile[getIoffset(tile)+i] then
20:         profile[getIoffset(tile)+i] ← privProfileI[i];
21:         profileIndex[getIoffset(tile)+i] ← privProfileIndexI[i];
22:       end if
23:       if privProfileJ[i] > profile[getJoffset(tile)+i] then
24:         profile[getIoffset(tile)+j] ← privProfileJ[i];
25:         profileIndex[getIoffset(tile)+j] ← privProfileIndexJ[i];
26:       end if
27:       privProfileI[i] ← −∞;
28:       privProfileJ[i] ← −∞;
29:     end for
30:     unsetlock(gLock);
31:   end for
32: end procedure
```

Figure 6: SCAMP private tiling lock-based algorithm pseudocode.

at the end of each metadiagonal. Although now the computational load of each tile is more or less the same (except for the tile in the last metacolumn), the ratio tiles/threads is now more important as the number of tiles in a metadiagonal decreases as we go further away from the main metadiagonal. Actually, the last metadiagonal has just one tile to be processed by one thread, with all other threads being idle. We discuss this in Sections 4.3.1 and 4.4.1.

4. Experimental Evaluation

4.1. Methodology

We used the experimentation server with the configuration depicted in Fig. 7. It comprises 4 sockets with Intel Xeon Gold 5218 at 2.30GHz [32]. Each processor has 16 cores sharing a 22MB L3 cache, with each core having two hyperthreads sharing two levels of private cache. The server allows for a total of 128 hardware threads executing in parallel.

We used a diverse set of time series for the experimental evaluation, whose parameters are shown in Table 1. We can see the number of samples (n) and the window size or subsequence length (m) used for each series, as well as the maximum and minimum values. The name of each series is descriptive of the

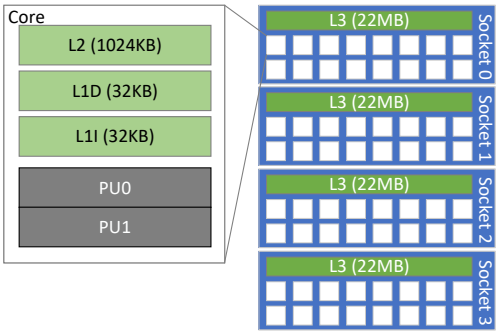

Figure 7: Intel Xeon Gold 5218 server configuration.

Table 1: Time series dataset parameterization

Time series	n	m	Max	Min	Up. %
Audio	20234	200	6.7	-56.5	0.27
ECG	180000	500	2.6	0.3	0.07
Human Activity	7997	120	2.5	-1.9	0.62
Penguin Behavior	109842	800	0.5	-0.2	0.18
Power	180000	1325	140	0	0.07
Seismology	180000	50	696.1	-185.7	0.01
ECG Large	1800000	500	3.4	-1.6	0.007
Power Large	1859587	1325	140	0	0.008
Seismo. Large	1728000	50	2329.3	-2334.6	0.002

type of content they contain. Note that all time series used in this work come from real data [33]. The last rows of the table name the series used to evaluate the proposals with large data sets. The last column shows the percentage of all profile update attempts that indeed write the matrix profile and matrix profile index arrays, using the baseline SCAMP algorithm with 1 thread, i.e. the if statements in Lines 16, 20, 28 and 32 in Fig. 4 which become true.

Regarding the performance results we considered six different implementation profiles:

- *Base*: SCAMP baseline algorithm with privatization and a final reduction stage to update from private to global profile (described in Section 3.1).
- *FGL*: SCAMP algorithm without privatization. Fine-grain locks are used to protect updates to the global profile (described in Section 3.2).
- *HTM*: SCAMP algorithm without privatization. HTM is used to protect updates to the global profile (described in Section 3.2).
- *TilesUnprot*: SCAMP tiling algorithm with privatization of the tile profile and a reduction stage without protection. This profile does not yield valid results and is used as a high performance ceiling reference for the following tiling profiles.
- *TilesLock*: SCAMP tiling algorithm with privatization of the tile profile and a reduction stage protected with a global lock (described in Section 3.3).
- *TilesHTM*: SCAMP tiling algorithm with privatization of the tile profile and a reduction stage protected with HTM (described in Section 3.3).
- *TilesMeta*: SCAMP tiling algorithm without privatization. Tiles in a metadiagonal are independent each other, so they are assigned to different threads that update the global profile directly. A barrier is required at the end of each metadiagonal (described in Section 3.4).

4.2. Transaction Size Sensitivity Analysis

Opening and closing transactions may have an associated overhead mainly to the fact that they imply memory barriers that orders all accesses caused by instructions before them ahead of all accesses caused by instructions after them [31]. Consequently, if we can decrease the number of transactions opened by the application, such an overhead will decrease as well.

In this case, decreasing the number of transactions implies making them larger (increasing XACT_SIZE, see Section 3.2). However, transactions cannot be made unlimited larger as the transactional system is best-effort and can overflow. For this reason, we carry out a transaction size sensitivity analysis where we used our HTM profile (described in Section 3.2) with 1 thread, so that no conflict among transactions of different threads can come up. Thus, the majority of transaction aborts are due to transactional hardware overflow, i.e. capacity aborts.

We propose the Capacity Abort Ratio (CAR) to find the optimum transaction size. CAR is calculated as:

$$CAR = \frac{Capacity\ Aborts}{Aborts + Commits}$$

where Aborts is the count of all the aborts reported by the transactional system, which can entail conflict aborts, capacity aborts, explicit aborts (those caused by the use of the XABORT instruction inside a transaction), and miscellaneous aborts (those caused by syscalls, interrupts, page faults,... found when executing a transaction) [34]. The point in which CAR begins to go up will mark the maximum size for transactions.

Fig. 8 shows the results for the transaction size sensitivity analysis. The figures have a very comprehensive set of metrics while varying the transaction size in the x-axis, which is given in terms of profile updates (see XACT_SIZE in Section 3.2). We show ratio bars with a breakdown of aborts (miscellaneous – Misc. –, explicit – Exp. –, conflict – Conf. –, and capacity – Cap. –) together with fallbacks; the CAR metric; and the speedup of the HTM profile with 1 thread with respect to the Base profile with 1 thread.

The results show a noticeable CAR increment for a transaction size of 512, especially for Audio and Human, which are the shortest series. These series show the highest update write percentage as shown in Table 1, which implies certain transactions has a larger footprint causing more capacity aborts.

On the other hand, the maximum speedup is obtained from transaction size 128 on wards, where it either stabilizes or gets down again from 512 on wards for some series. The ratio of aborts in the bars goes from having barely no capacity and all miscellaneous aborts for Human and transaction size 1 to almost all capacity aborts in every series from 512 transaction size on wards. Other series shows a high capacity abort figure on the bars with transaction size 1, but we have to take into

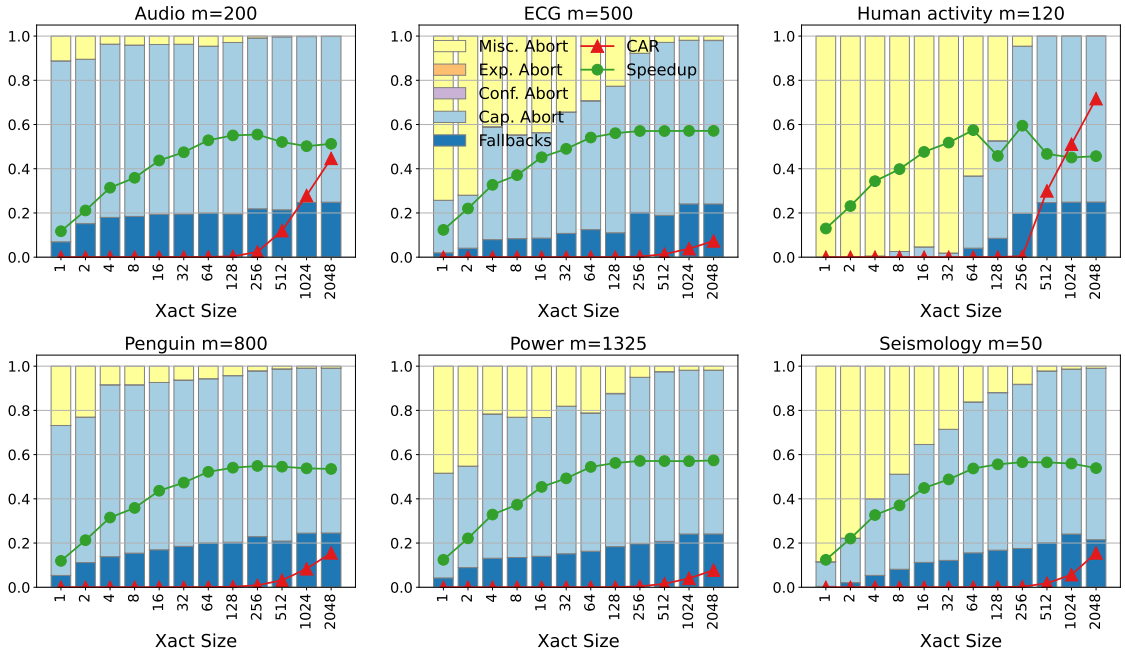

Figure 8: Transaction footprint sensitivity with 1 thread for the HTM profile. CAR and speedup with abort rate breakdown.

account that this configuration starts a huge amount of transactions and the number of aborts is very low, and although the transactions are small they can show capacity aborts because of a cache set replacement of a transactional block.

As a consequence of the sensitivity analysis, we choose a XACT_SIZE of X=128 for our experiments, as we get a speedup very similar to that yielded by the 256 configuration, and we have to take into account that hyperthreading configurations will have two threads sharing the transactional hardware of the core. This way, there will be less pressure to the transactional system. Transaction size of 512 is discarded as it is the point from which the CAR begins to rise dramatically.

4.3. Results

Once decided the transaction size for the HTM and TilesHTM profiles, we conduct a series of experiments with all the profiles described in Section 4.1. The results yielded by these experiments are depicted in Fig. 9, where each row of plots belongs to a series (except for the large ones, which are discussed in Section 4.4), and each column to a tile size, L, instantiated to 128, 512, 2058 and 8192. This way, the results for the Base, FGL, and HTM profiles are the same for different columns of a given row, whereas Tiles-profiles' results change. Note that the y-axis shows speedup with respect to the Base profile with 1 thread on the interval [0, 45] for all plots. The x-axis is the number of threads spawned by the application in the range from 1 to 128 in binary powers.

At first glance, we can observe that the **Base** profile peaks close to 30× speedup with 64 threads for all series but the shorter ones, i.e. Audio and Human, which get 13× and 7× respectively. We can also see a slowdown with 128 threads for all series. Although we can thought of hyperthreading as responsible for such a loss of performance, neither large series (see Fig. 10) nor tile profiles show this behavior, which leads to blame load imbalance. The thread's work unit in the Base profile is the distance matrix diagonal, and it gets shorter as we get further away from the main diagonal, to such an extent that the last diagonal comprises just one distance calculation (see Fig. 2). Having many threads and not so many diagonals to compute makes this imbalance evident, and for Audio and Human, with less than 20K diagonals to compute, it even affects the overall speedup.

When it comes to the **FGL** profile, each thread works on the global arrays (time series, matrix profile, and matrix profile index) without privatization. An array of locks is declared with a lock per element of the matrix profile, to protect the updates of each element of the matrix profile and matrix profile index arrays. This means that there are two additional memory accesses per update to acquire and release the lock, which is the main source of slowdown for this profile. The incremental computation of the remainder of a diagonal from its first distance value is bound by DRAM memory roof (see Motivation in [35]), so adding more accesses per operation can saturate the memory bandwidth. Fig. 9 shows the FGL profile barely going up 3× speedup with 64 threads.

The transactional counterpart of FGL, the **HTM** profile, also works on the global arrays without the need for privatization, but in this case the updates are protected with transactions (with size X=128 as derived from Section 4.2) so locks are not needed either. The results yield a substantial performance gain of HTM over FGL, with speedup peaks among 10× and 20× for 64 threads. The optimistic nature of transactions (there is no serialization unless necessary, in case of a conflict) and the fact that a small fraction of updates actually write the profile (see Table 1) makes HTM outperform FGL. However, Audio and Human short series has an increased percentage of update writes

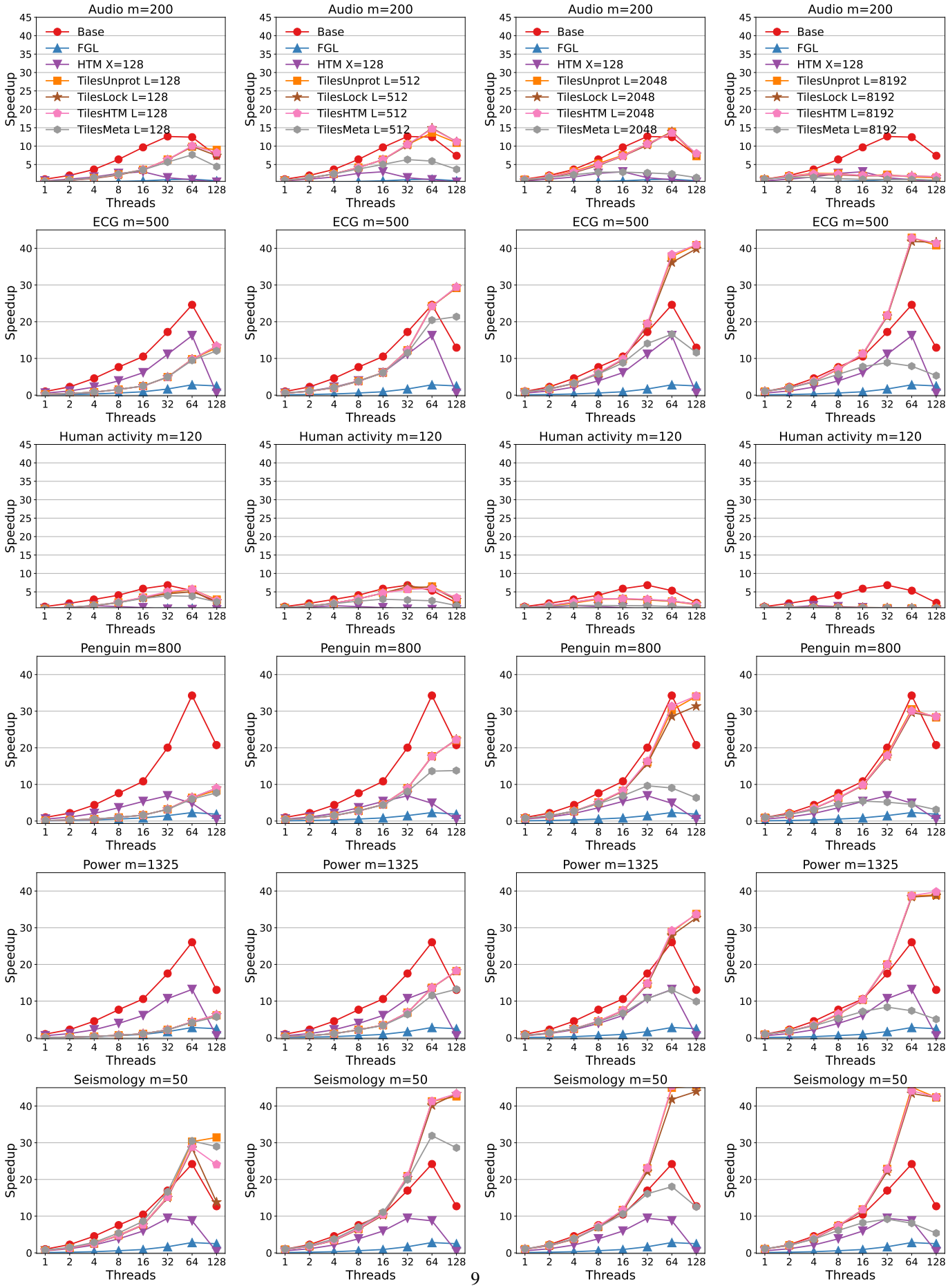

Figure 9: Speedup results.

which penalizes these series (e.g. Human yields a 56% of conflict aborts, while Seismology only a 28% for 128 threads), and opening and closing transactions do not come without a cost, so the HTM profile never beats the Base profile. For 128 threads we can observe an increased slowdown with respect to Base, that in this case is due to hyperthreading sharing transactional resources among hardware threads (capacity aborts can raise up to two orders of magnitude from 64 threads to 128).

4.3.1. Tiling results

The results of the tiling profiles depend on the tile size but, overall, they yield excellent performance gains, as shown in Fig. 9, because of data reuse (same data is used to compute the lower and the upper triangle of a tile) and better load balance (tiles are the same size). We evaluate four tiling profiles: TilesUnprot is like TilesLock and TilesHTM with tile privatization and final reduction, but without protection, so it is not correct but it is intended to serve as a performance upper bound. TilesLock uses a global lock to perform the reduction and TilesHTM uses transactions of size 128. TilesMeta does not need tile privatization but it uses a barrier per metadiagonal. The tile size, L, i.e. the number of time series elements comprising the tile edge, goes from 128 to 8K, left to right.

When it comes to tiling, another variable comes in scene in addition to the tile size: the subsequence length, m. Unlike Base, FGL, and HTM, which calculate the covariance and correlation between the two first subsequences of the diagonal and the rest is calculated incrementally from that value, the tiling profiles do the same within the tile but have to compute the covariance and correlation again from scratch when moving to other tile. The smaller the tile the more tiles to compute and the greater the increase in calculations, especially when the subsequence length is large. With **L=128**, the leftmost column of plots, we have the smallest tile and, consequently, the greater increase in calculations with respect to the Base, FGL, and HTM profiles. Thus, tiling profiles yield worse results in this case for all series except for Seismology, which has the lowest subsequence length of $m = 50$ and such a calculation overload is compensated by the fact that tiling profiles shows better load balance as the thread work unit is the tile. With **L=512** we can see that tiling profiles get better, but they do not beat the Base yet until we spawn 128 threads because of the load balance. With **L=2048** and **L=8192** the tiling profiles outperform the Base profile in most cases with around 40× speedup over 1 thread and between 2× and 3× speedup over Base and 128 threads. The shortest series, Audio and Human, do not yield any profit as the tile size increases since threads fall short of tiles to compute.

As far as the tile reduction protection method is concerned, there is not much of a difference between using locks or HTM. In fact, the **TilesLock** and **TilesHTM** profiles perform the same as the **TilesUnprot** profile, which implies that threads synchronize their tile computation stage (on private arrays and floating computation demanding) and reduction stage (protected critical section to update the global profile) in a way they do not have to wait for each other. Actually, the only scenario where we can spot a difference among these profiles is in Seismology m=50 with L=128 and 128 threads. With L=128 we have the greater amount of tiles, and therefore, the greater amount of reduction critical sections. As m=50 and L=128, the tile computation stage is not so heavy with respect to the reduction stage, so the latter has more impact on performance. With 128 threads, the time waiting at a global lock to perform the reduction is greater than that of the tile computation. Then, TilesUnprot outperforms the rest, and TilesHTM beats the TilesLock profile as this series shows a very low update percentage (see Table 1) so the optimistic concurrency provided by HTM enhances parallelism.

The **TilesMeta** profile performs the same as the other tiling profiles when there is enough tiles to share among threads within a metadiagonal, i.e. for low tile sizes. When L is greater, there are enough tiles for threads to work within a metadiagonal and load imbalance shows up, getting worse than the other profiles as tile size increases. Next section gains more insight into this profile.

4.4. Large series results

The results for large series are shown in Fig. 10. The length of these series is around 1.8 million samples (see Table 1), an order of magnitude greater than the series in Fig. 9. We do not carry out the experiments with large series for FGL because of its poor performance.

The **Base** profile gets similar results as with short series up to 64 threads. However, it now scales wonderfully with 128 threads reaching 50× speedup over 1 thread. Having so many diagonals to share among so many threads gives more chance for them to load balance properly.

However, we can see that **HTM** does not scale well with the size of the time series, as this profile yields up to 6× speedup for large series, whereas it peaked 10× with short series. Although n is one order of magnitude greater now, the number of transactions increases by two orders of magnitude, and so it does the transactional overhead (more conflicts, more aborts, open/close transaction overhead,...).

4.4.1. Tile size heuristic

As far as tiling profiles are concerned with large series, we obtain similar results as with short series except for the TilesMeta profile. That is, **TilesLock** and **TilesHTM** perform the same as **TilesUnprot**, but for Seismology and L=128 and 128 threads, where TilesHTM beats TilesLock for the reasons exposed in the last section. The tiling profiles get up to 80× speedup over the Base with 1 thread and between 2× and 3× better performance than the Base profile. Note the super-linear speedup for Seismology, up to 64 threads and L≥512, probably due to the added cache and the fact that the i and j elements of the upper triangle of a tile are reused in the computation of the lower tile, and in the case of Seismology there is no much time wasted recomputing the first correlation and covariance as m=50.

After reviewing the results of large and short series, we can come up with an heuristic (Eq. 9) for the tile size and the number of threads needed to outperform the Base profile. The number of elements in a triangular matrix counting the elements in

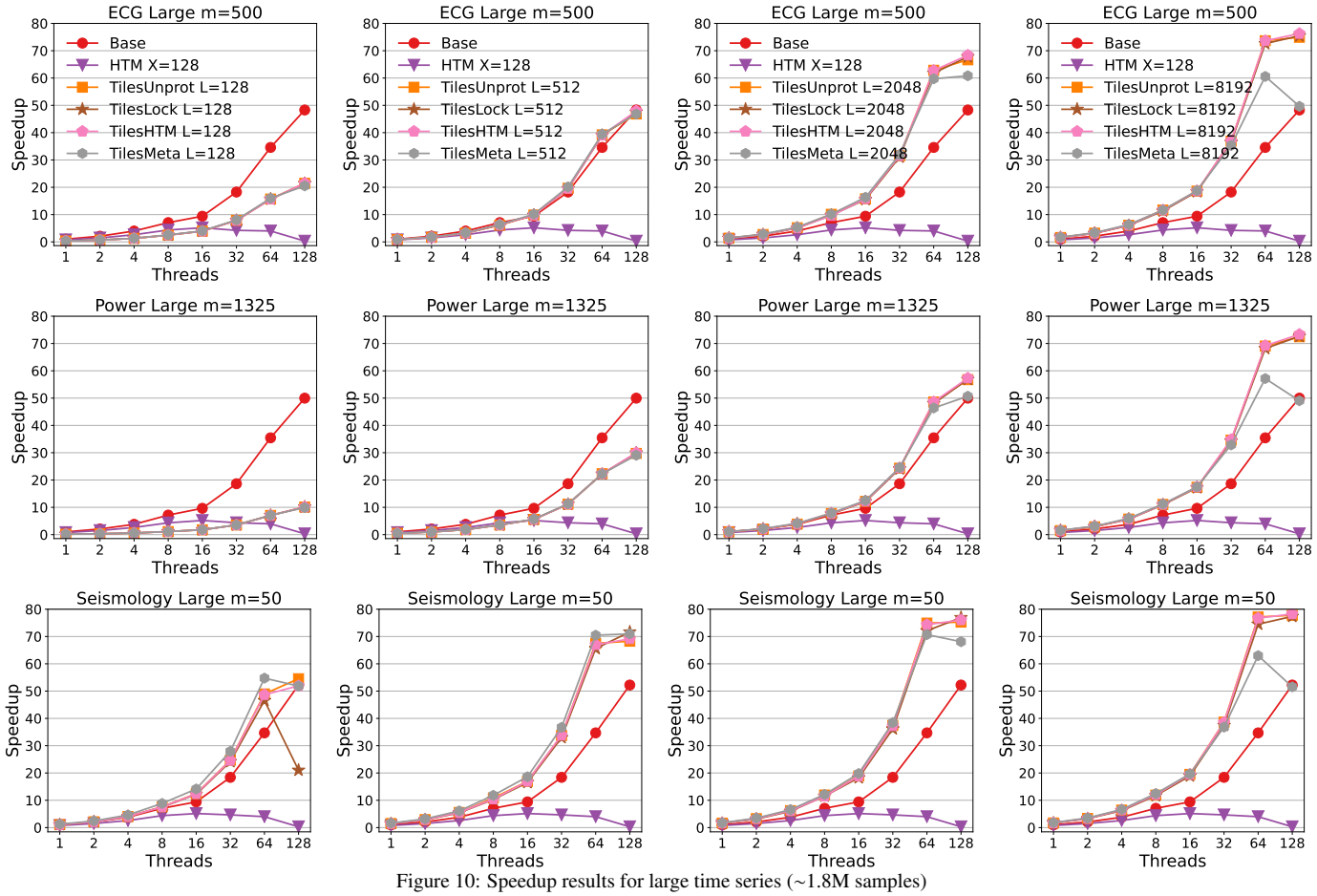

Figure 10: Speedup results for large time series (~1.8M samples)

the diagonal can be expressed as $N(N+1)/2$, being the number of tiles in the first metarow $N = \lceil l/L \rceil$, with $l = n - m + 1$. This way, the overhead of recomputing the covariance and correlation of the first elements of each tile compensates with the gains of load balance.

$$L \geq m \quad \wedge \quad thCount \leq \lceil l/L \rceil (\lceil l/L \rceil + 1)/2 \tag{9}$$

The **TilesMeta** profile will always have load imbalance when approaching the top-right corner of the matrix of tiles, where the number of tiles in the metadiagonal is lower than the number of threads (see Fig. 5). However, when the ratio l/L is much larger than the number of threads, the load imbalance gets negligible. For that reason, TilesMeta now performs very similar to TilesUnprot, TilesLock, and TilesHTM up to L=2048, whereas we noticed a slowdown from L=512 on with short series. We can now notice such a slowdown with large series and L=8192 and 128 threads, as the ratio l/L is very close to 128 and load imbalance hinders the overall performance of the profile.

4.5. Memory Requirements

Memory requirements for time series mining algorithms can become a major issue when dealing with very large time series [23, 24, 25]. Matrix Profile already optimizes the memory footprint by only storing the matrix profile and matrix profile index arrays instead of the entire distance matrix. However, its memory requirements can increase because of privatization depending on the number of threads. In this section, we discuss the impact on memory footprint of the explored profiles.

Table 2 shows the memory requirements in MB for the different profiles when computing the large series with 128 threads. We break down the memory required by the implementation profile into memory space needed by the time series, T, the statistics, and the matrix profile, P, and the matrix profile index, I. These arrays are of types double and long integer, which comprise both 8 bytes per element in a 64bit architecture as the one described in Section 4.1. The Total column adds up the size of all arrays.

We can observe that the Base profile is the most bulky in terms of memory. This is because of the need for privatizing P and I arrays per thread (128 in this case), plus the global ones. It can take 3.5GB of memory to compute a series of 1.8 million samples. Imaging dealing with series one or two orders of magnitude larger. The memory requirements would be unapproachable.

The HTM profile, though, reduces the memory needs to the maximum as it works with the global arrays P and I. It is worth noting that the memory requirements do not depend on the number of threads in this case. On the other hand, we get the HTM system performance penalty. In case of short series and up to 64 threads (where transaction contention is not so apparent) we could use this profile if memory constraints are hard.

Table 2: Memory requirements with 128 threads for the large series and the different profiles.

		Size (MB)			
Time series	**Profile**	**T**	**Statistics** (μ, σ, df, dg)	**P,I**	**Total**
ECG Large	Base	13.73	54.92	3542.11	3610.76
	HTM X=128	13.73	54.92	27.46	96.11
	TilesMeta L=8192	13.73	54.92	27.46	96.11
	TilesLock L=8192	13.73	54.92	59.46	128.11
	TilesHTM L=8192	13.73	54.92	59.46	128.11
Power Large	Base	14.19	56.71	3657.77	3728.67
	HTM X=128	14.19	56.71	28.35	99.25
	TilesMeta L=8192	14.19	56.71	28.35	99.25
	TilesLock L=8192	14.19	56.71	60.35	131.25
	TilesHTM L=8192	14.19	56.71	60.35	131.25
Seismology Large	Base	13.18	52.73	3401.27	3467.19
	HTM X=128	13.18	52.73	26.37	92.28
	TilesMeta L=8192	13.18	52.73	26.37	92.28
	TilesLock L=8192	13.18	52.73	58.37	124.28
	TilesHTM L=8192	13.18	52.73	58.37	124.28

The best profiles are the ones using tiling, not only in performance but also when it comes to memory. TilesLock and TilesHTM require more memory than TilesMeta, around 25% more, because of the need for privatizing P and I for the tile (we need P and I arrays, of size L, for the i and j dimensions of the tile). It is not a noticeable increment, especially when looking at the requirements of the Base profile. But it can be a problem as the tile size, L, and the number of threads increases. The memory requirements for these profiles depends on such parameters. In exchange, we can get the best performance results.

TilesMeta has the same memory requirements as the HTM profile, and neither depends on the number of threads nor the tile size. Moreover, it gets very good performance results for large series when the l/L ratio is greater than the number of threads.

5. Related Work

Matrix Profile multicore parallelization relies on several cores updating their private copies of P and I and a final stage to merge the results into the global arrays [10]. Although they claim the space complexity is $O(n)$ [26], beating other proposals as [24, 36], it actually depends on the parallelization method, being $O(n + l \cdot thCount)$ for multiprocessor architectures, where $l = n-m+1$. If $m \ll n$, which is usually the case with real world problems having m set to thousands of samples [26, 27], we end up with space complexity $O(n \cdot thCount)$. Our proposals are able to reduce such a memory footprint to $O(n + L \cdot thCount)$ space complexity for TilesHTM, with $L \ll n$, and $O(n)$ for TilesMeta, while being 2× to 3× faster than the baseline.

As the Matrix Profile algorithm is highly parallelizable they provide GPU implementations to harness SIMD parallelism [26, 22]. In [26] they propose to copy the global arrays into GPU's private memory, computing the first row of the distance matrix and having all GPU threads incrementally computing from that the rest of the rows of the matrix in parallel (each thread computes an element of the row following a diagonal, and updates the global P and I arrays, having $O(n)$ space complexity). As D is a triangular matrix, load imbalance becomes more apparent as the process approaches the last rows. Actually, they mention that the CPU should provide equal amount of work to each GPU in the case of having a GPU farm (2 GPUs are used in their experiments). They claim that a partition of D is assigned to each GPU maintaining $O(n)$ space complexity, but they do not mention the need for having private P and I arrays.

In [22] they improve the GPU approach by using tiling and having each tile assigned to a GPU from the GPU farm. They do mention now the need for maintaining P and I private arrays per tile and updating the global arrays with atomic instruction. To do so, they impose the hard constraint of having a single precision floating value for the profile and a 32bit integer index packed together, so that they can be atomically updated at a time with atomic intrinsics operating on 64bit words. They also work with tile sizes of about 1 million samples (with its implication on memory footprint) to fully saturate each GPU worker. We have used a similar tiling approach for our multicore tiling approaches, but we do not restrict the type of P and I arrays (actually we used 64bit floats and integers for our experiments) and the tile size can be set arbitrarily, obtaining great results with low L values that harness the cache hierarchy, both on speedup and memory footprint.

The work [23] highlights the problem of time series analysis algorithms either performing approximate motif finding or considering relatively small datasets residing in main memory. They propose an algorithm to work with disk-resident data sorting it to optimize the number of disk block accesses, and do not consider parallelization. Our proposals focus on optimizing performance and main memory footprint of in-memory exact motif discovery algorithms using parallelization.

Prätzlich et al. [25] approach the problem of global sequence alignment, i.e. finding how similar two entire time series are each other, which is different from our approach of finding motifs and discords in time series. They use the Dynamic Time Warping (DTW) method between two entire time series reducing its space complexity by computing the DTW in small blocks and computing these blocks in parallel, which inspired our tiling multiprocessor proposals.

6. Conclusions

In this work we explore several multiprocessor implementation approaches to a Matrix Profile time series analysis algorithm, evaluating both speedup and memory requirements, the latter being a major issue when dealing with very large time series.

We propose new implementations using Hardware Transactional Memory (HTM) extensions provided by several multiprocessors nowadays, and we also resource to tiling to enhance the load balance and cache reuse of the algorithm. As a result, we obtain a tiling HTM approach which outperforms the baseline implementation in most cases and shows reduced memory requirements. We come up with a heuristic to choose the proper tile size and thread count depending on the subsequence and time series lengths, so that the proposed implementation ensures better performance than the baseline. We also propose a novel tiling implementation without the need for synchronization other than barriers, that requires the lowest memory requirements and performs remarkably when analysing very large time series.

Our proposals show speedups of up to 70× and 80× over sequential with 128 threads, and memory requirements not depending on the number of threads spawned by the application.

Acknowledgments

Governments of Spain (project TIN2016-80920-R) and Junta de Andalucía (project P12-TIC-1470).

References

[1] R. H. Shumway, D. S. Stoffer, Time Series Analysis and Its Applications, 4th Edition, Springer-Verlag, 2017.

[2] B. Chiu, E. Keogh, S. Lonardi, Probabilistic Discovery of Time Series Motifs, in: Int'l Conf. on Knowledge Discovery and Data Mining (KDD'03), 2003, pp. 493–498.

[3] E. Keogh, J. Lin, S.-H. Lee, H. Van Herle, Finding the Most Unusual Time Series Subsequence: Algorithms and Applications, Knowl. Inf. Syst. (2006).

[4] A. McGovern, D. H. Rosendahl, R. A. Brown, K. K. Droegemeier, Identifying predictive multi-dimensional time series motifs: An application to severe weather prediction, Data Mining and Knowledge Discovery (2011).

[5] C. Cassisi, M. Aliotta, A. Cannata, P. Montalto, D. Patanè, A. Pulvirenti, L. Spampinato, Motif discovery on seismic amplitude time series: The case study of Mt Etna 2011 eruptive activity, Pure Appl. Geophys. (2013).

[6] B. Szigeti, A. Deogade, B. Webb, Searching for motifs in the behaviour of larval Drosophila melanogaster and Caenorhabditis elegans reveals continuity between behavioural states, Journal of The Royal Society Interface (2015).

[7] P. Garrard, V. Nemes, D. Nikolic, A. Barney, Motif discovery in speech: Application to monitoring alzheimer's disease, Current Alzheimer Research (2017).

[8] S. Torkamani, V. Lohweg, Survey on time series motif discovery, Wiley Interdis. Rev. (WIREs) Data Mining and Knowledge Discovery (2017).

[9] E. Cartwright, M. Crane, H. J. Ruskin, Financial Time Series: Motif Discovery and Analysis Using VALMOD, in: Int'l. Conf. on Computational Science (ICCS'19), 2019, pp. 771–778.

[10] C.-C. M. Yeh, Y. Zhu, L. Ulanova, N. Begum, Y. Ding, H. A. Dau, D. F. Silva, A. Mueen, E. Keogh, Matrix Profile I: All pairs similarity joins for time series: A unifying view that includes motifs, discords and shapelets, in: Int'l Conf. on Knowledge Discovery and Data Mining (KDD'16), 2016, pp. 1317–1322.

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

[12] N. Shavit, D. Touitou, Software transactional memory, in: Symp. on Principles of Distributed Computing (PODC'95), 1995, pp. 204–213.

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

[14] D. Dice, O. Shalev, N. Shavit, Transactional locking II, in: Int'l. Symp. on Distributed Computing (DISC'06), 2006, pp. 194–208.

[15] P. Felber, C. Fetzer, P. Marlier, T. Riegel, Time-based software transactional memory, Trans. on Parallel and Distributed Systems 21 (12) (2010) 1793–1807.

[16] L. Hammond, V. Wong, M. Chen, et al., Transactional memory coherence and consistency, in: Int'l. Symp. on Computer Architecture (ISCA'04), 2004, pp. 102–113.

[17] R. Rajwar, M. Herlihy, K. Lai, Virtualizing transactional memory, in: Int'l. Symp. on Computer Architecture (ISCA'05), 2005, pp. 494–505.

[18] K. Moore, J. Bobba, Moravan, et al., LogTM: Log-based transactional memory, in: Int'l. Symp. on High-Performance Computer Architecture (HPCA'06), 2006, pp. 254–265.

[19] C. Kevin Shum, F. Busaba, C. Jacobi, IBM zEC12: The third-generation high-frequency mainframe microprocessor, Micro 33 (2) (2013) 38–47.

[20] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, T. Burton, Haswell: The Fourth-Generation Intel Core Processor, Micro 34 (2) (2014) 6–20.

[21] H. Q. Le, G. L. Guthrie, D. E. Williams, et al., Transactional memory support in the IBM POWER8 processor, IBM Journal of Research and Development 59 (1) (2015) 8:1–8:14.

[22] Z. Zimmerman, K. Kamgar, N. S. Senobari, B. Crites, G. Funning, P. Brisk, E. Keogh, Matrix profile XIV: Scaling time series motif discovery with GPUs to break a quintillion pairwise comparisons a day and beyond, in: Symp. on Cloud Computing (SoCC'19), 2019, pp. 74–86.

[23] A. Mueen, E. Keogh, N. Bigdely-Shamlo, Finding Time Series Motifs in Disk-Resident Data, in: Int'l. Conf. on Data Mining (ICDM'09), 2009, pp. 367–376.

[24] Y. Li, H. U. Leong, M. L. Yiu, Z. Gong, Quick-motif: An efficient and scalable framework for exact motif discovery, in: Int'l. Conf. on Data Engineering (ICDE'15), 2015, pp. 579–590.

[25] T. Prätzlich, J. Driedger, M. Müller, Memory-restricted multiscale dynamic time warping, in: Int'l. Conf. on Acoustics, Speech and Signal Processing (ICASSP'16), 2016, pp. 569–573.

[26] Y. Zhu, Z. Zimmerman, N. S. Senobari, C.-C. M. Yeh, G. Funning, A. Mueen, P. Brisk, E. Keogh, 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.

[27] Y. Zhu, C.-C. M. Yeh, Z. Zimmerman, K. Kamgar, E. Keogh, Matrix Profile XI: SCRIMP++: Time series motif discovery at interactive speeds, in: Int'l Conf. on Knowledge Discovery and Data Mining (KDD'18), 2018, pp. 837–846.

[28] M. M. K. Martin, C. Blundell, E. Lewis, Subtleties of Transactional Memory Atomicity Semantics, Comput. Archit. Lett. 5 (2) (2006) 17.

[29] M. J. Moravan, J. Bobba, K. E. Moore, et al., Supporting Nested Transactional Memory in logTM, in: Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06), 2006, pp. 359–370.

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

[31] H. Q. Le, G. L. Guthrie, D. E. Williams, M. M. Michael, B. G. Frey, W. J. Starke, C. May, R. Odaira, T. Nakaike, Transactional memory support in the IBM POWER8 processor, IBM Journal of Research and Development 59 (1) (2015) 8:1–8:14.

[32] Procesador intel® xeon® gold 5218, `https://ark.intel.com/content/www/xl/es/ark/products/192444/intel-xeon-gold-5218-processor-22m-cache-2-30-ghz.html`, accessed july (2023).

[33] The UCR Matrix Profile Page, `https://www.cs.ucr.edu/~eamonn/MatrixProfile.html`, accessed july (2023).

[34] Intel 64 and IA-32 Architectures Optimization Reference Manual. Chapter 16 Intel® TSX Recommendations (Jan 2023).
[35] R. Quislant, I. Fernandez, E. Gutierrez, O. Plata, Time series analysis acceleration with advanced vectorization extensions, The Journal of Supercomputing 79 (9) (2023) 10178–10207.
[36] A. Mueen, E. Keogh, Q. Zhu, S. Cash, B. Westover, Exact discovery of time series motifs, in: SIAM Int'l. Conf. on Data Mining (SDM'09), 2009, pp. 473–484.