Project Final Report: Matrix Profile MPI Implementation

Brody Larsen* and Richard McNew†
Department of Computer Science, Utah State University
Logan, Utah, USA
*a01977457@usu.edu, †a02077329@usu.edu

***Abstract*—The Matrix Profile is an amazing data structure and set of accompanying algorithms that have revolutionized time series data mining tasks over the past five years. Most data science work is done in Python. As a result, most Matrix Profile implementations in use today are written in Python and rely on the NumPy, SciPy, and Numba Python libraries for vector and matrix data types, numerical and scientific algorithms, and fast just-in-time optimizations. In this project we created an MPI implementation of the Matrix Profile in C++.**

***Index Terms*—parallel computation, MPI, data science, Matrix Profile, C++**

I. Introduction

Time series data is a collection of observations made sequentially in time. Time series data is ubiquitous in our modern world and takes the form of sensor data, stock market data, network metrics, application logs, and many other forms. Analyzing time series data is essential to understanding what is happening and enables informed decision-making and strategic planning. One recent advancement in time series data analysis is the Matrix Profile[1].

The Matrix Profile is an amazing data structure and set of accompanying algorithms that annotate a time series and make most time series data mining problems easy to solve[2]. The Matrix Profile is: 1) exact - it allows for time series analysis without false positives or false negatives, 2) parameter-free - unlike many time series data analysis tools, no hyperparameter tuning is needed, 3) space efficient - a matrix profile data structure does not require much space, enabling large datasets to be processed in memory, 4) parallelizable - it is fast to compute on modern hardware, and 5) simple - it is easy to use and fairly easy to understand[7].

Most data science work is done in Python. As a result, most Matrix Profile implementations in use today are written in Python[6] and rely on the NumPy, SciPy, and Numba Python libraries for vector and matrix data types, numerical and scientific algorithms, and fast just-in-time optimizations. Creating a Matrix Profile implementation using MPI in C++ will offer organizations that use MPI a way to use the Matrix Profile.

II. Project Thesis

This project created an MPI implementation of the Matrix Profile in C++.

III. Background

A. Definitions and Notation

We must first formally define a *time series*:

Definition 1: A *time series* T is a sequence of real-valued numbers t_i: $T = t_1, t_2, \ldots, t_n$ where n is the length of T.

In most cases, only some parts of the time series are interesting rather than the entire time series. These *subsequences* are what we want to find and study.

Definition 2: A *subsequence* $T_{i,m}$ of a *time series* T is a continuous subset of the values of T of length m starting from position i. Formally, $T_{i,m}$ = t_i, t_{i+1}, …, t_{i+m-1}, where $1 \leq i \leq n - m + 1$.

Given a query subsequence $T_{i,m}$ and a time series T, we can compute the distance between $T_{i,m}$ and *all* the subsequences in T. We call this a *distance profile*:

Definition 3: A *distance profile* D_i corresponding to a query subsequence $T_{i,m}$ and a time series T is a vector of the Euclidean distances between the given query subsequence $T_{i,m}$ and each subsequence in the time series T. Formally, D_i = $[d_{i,1}, d_{i,2}, \ldots, d_{i,n-m+1}]$, such that $d_{i,j} (1 \leq j \leq n - m + 1)$ is the distance between $T_{i,m}$ and $T_{j,m}$.

Once D_i is calculated it can be used to find the nearest neighbor of $T_{i,m}$ in T. If the query subsequence $T_{i,m}$ is a subsequence of T (itself, as opposed to another time series), then the i^{th} location of the distance profile D_i will be zero (that is, $d_{i,i} = 0$) and almost zero just to the left and right of i. This is called a *trivial match* since that point in the time series matches itself exactly and thus must be ignored. To ignore trivial matches, an "exclusion zone" of length m/k before and after i (the location of the query) is used, where $1 < k < m - 1$. Emprically, an exclusion zone of $m/4$ works well for most applications. Figure 1 shows how different sizes of exclusion zones impact the distance profile construction[6].

We want to find the nearest neighbor for every subsequence of T. The nearest neighbor information is store in two meta time series, the *Matrix Profile* and the *Matrix Profile Index*:

Definition 4: A *Matrix Profile* P of time series T is a vector of the Euclidean distances between every subsequence of T and its nearest neighbor in T. Formally, P = [min(D_1), min(D_2), …, min(D_{n-m+1})], where $D_i (1 \leq i \leq n-m+1)$ is the distance profile D_i corresponding to the query subsequence $T_{i,m}$ and time series T.

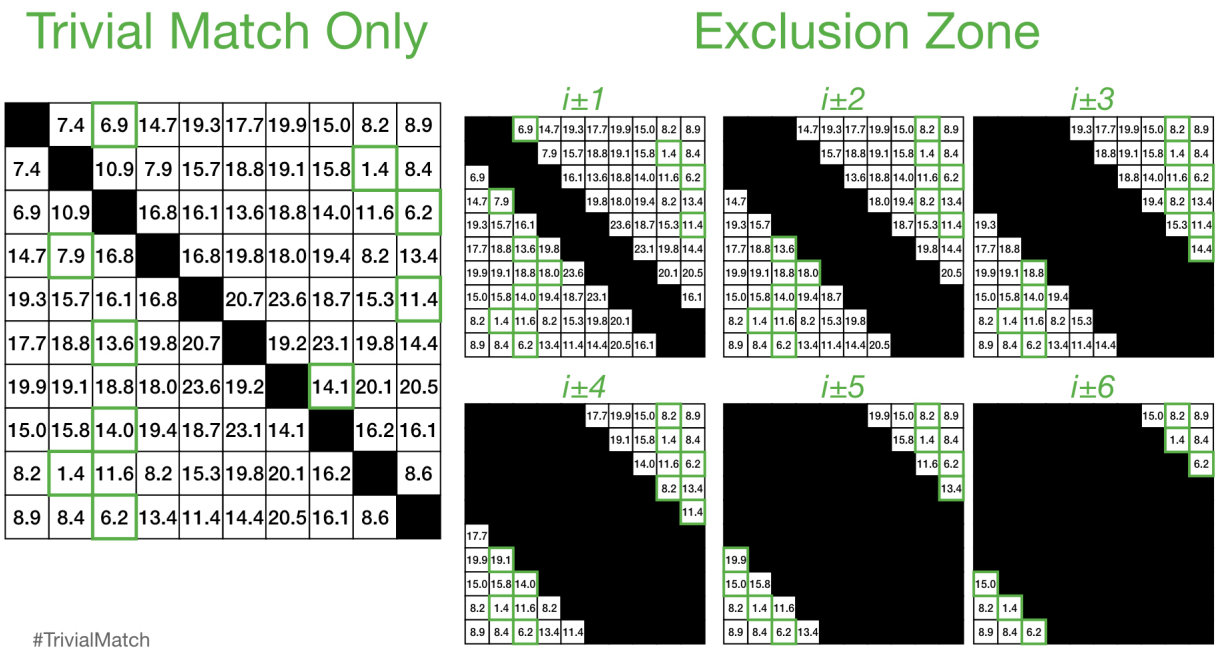

Fig. 1. Exclusion Zone Sizes

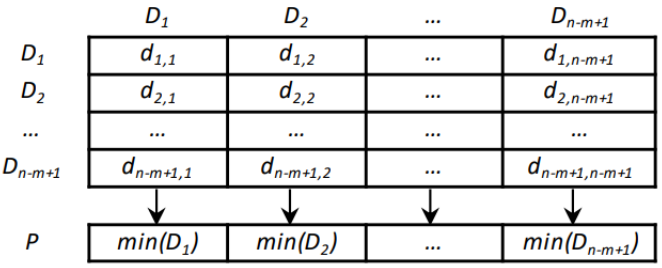

Fig. 2. Distance Profile Minima Collected into Matrix Profile

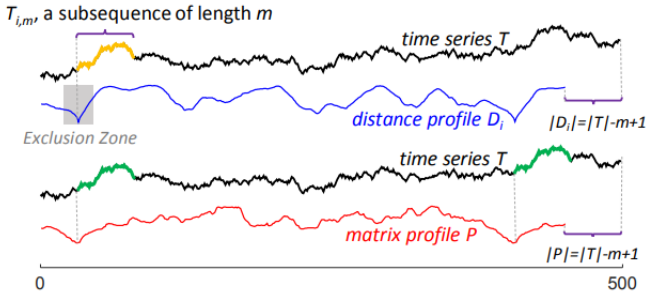

Fig. 3. Definitions Illustrated

The i^{th} element of the Matrix Profile gives the Euclidean distance from the query subsequence $T_{i,m}$ to its nearest neighbor in time series T. However, it does not give the *location* of that nearest neighbor; this information is stored in the *Matrix Profile Index*:

Definition 5: A *Matrix Profile Index* I of time series T is an integer vector: $I = [I_1, I_2, \ldots, I_{n-m+1}]$, where $I_i = j$ if $d_{i,j}$ = min(D_i).

Figure 2 illustrates the relationship between distance profiles and the Matrix Profile. Each row in the distance matrix is a distance profile. The minimum of each column is selected and collected in the Matrix Profile. (The diagonal and the exclusion zone immediately around it are ignored.) The Matrix Profile Index stores the location where the minimum of that column was found[4].

Each distance matrix element $d_{i,j}$ is the distance between $T_{i,m}$ and $T_{j,m}(1 \leq i, j \leq n - m + 1)$ of time series T. Figure 3 shows an example time series and how the above definitions might be visualized[3]. These definitions present the Matrix Profile as a *self-join*. That is, for every subsequence in a time series T, we find its non-trivial-match nearest neighbor within the *same* time series. To be more broadly useful, the Matrix Profile can be extended to be an AB-join: for every subsequence in a time series A, find the nearest neighbors in time series B [2][4]. For the purposes of this project, we will only deal with the self-join scenario.

IV. Approach

The Matrix Profile has revolutionized time series data mining tasks and has been heavily optimized, improved, and enhanced since it was announced in 2016. At the time of this project (2022), there are twenty-three Matrix Profile

papers[7] and many Matrix Profile algorithms: STAMP, STAMP*I*, STOMP, SCRIMP, SCRIMP++, SWAMP, and GPU-STOMP. This project uses the original STAMP algorithm described in the first Matrix Profile paper[1] with an aim for simplicity in implementation.

The MPI implementation strategy for Matrix Profile is fairly simple for time series data that will fit in the working memory of the MPI processor computers.

1) Leader (rank 0) MPI process parses and validates command line arguments to get an input time series filename, an output matrix profile filename, a subsequence window size, and an input time series column (time series data is frequently stored in comma-separated value formats).
2) Leader MPI process reads in time series data from the specified input file.
3) Leader MPI process `MPI_Bcasts` time series data to all other processes.
4) Each MPI process works on distance profile calculations for a segment of time series indices (values of i) based on their MPI process rank. For example, if there are 4 MPI processes, then rank 0 will work on the first fourth of the indices, rank 1 will work on the second fourth of the indices and so forth.
5) Each MPI process maintains its own local Matrix Profile and local Matrix Profile Index consisting of the minima and indices of the distance profiles it calculates. Each time a distance profile is calculated, it is merged into the local Matrix Profile by performing an element-wise minimum with the exception of the trivial match and the surrounding exclusion zone.
6) After each non-Leader MPI process finishes its segment of time series indices, it `MPI_Sends` its local Matrix Profile and local Matrix Profile Index to the Leader (rank 0) process.
7) After the Leader process finishes its segment of the time series indices, it `MPI_Recvs` the local Matrix Profiles and local Matrix Profile Index from the other processes. For each received local Matrix Profile and local Matrix Profile Index, the Leader process merges the received Matrix Profile and Index into its own using the same element-wise minimum algorithm used to merge a distance profile into the local Matrix Profile.
8) Once all the other processes' local Matrix Profiles and Indices have been merged, the Leader process writes the final Matrix Profile to the output file.

V. Simulation

There are no simulations for this project.

VI. Tests

In order to validate the finished MPI implementation of Matrix Profile in C++ a test suite was created. The test suite contains nine input time series and their corresponding matrix profiles. The STUMPY[6] Python library implementation of the Matrix Profile was used to process the input time series and the output Matrix Profile data structures were captured. The MPI Matrix Profile implementation was validated as correct because the output Matrix Profile data structures match those created by the STUMPY Matrix Profile implementation.

Additionally, unit tests were created to validate that each component of the MPI Matrix Profile implementation works as expected. GitHub Actions were also used to perform multi-platform (Linux, MacOS, and Windows) builds and unit tests before pull requests were allowed to merge into the `main` branch. See https://github.com/rmcnew/parallel_group_project/actions for more information.

VII. Test Datasets

Real world time series data was used in the test suite. Figure 4 gives time series test datasets input files and a brief description of each. Note that the time series used are relatively small compared to some real world time series data which range up to multiple gigabytes in size for rapidly generated or multivariate data. The number of datapoints for each time series is given to show its size.

VIII. Results

The MPI Matrix Profile implementation calculated the same results as the STUMPY Matrix Profile implementation with 99.95% similarity. Figure 5 lists the percent similarity for each input time series matrix profile.

The percent similarity calculations were performed as the inverse of the percent difference: $percent\_similarity = \left(1 - \frac{|actual - expected|}{expected}\right) * 100$ where the $expected$ values were those provided by the STUMPY matrix profile output and the $actual$ values were those provided by our MPI matrix profile output. Note that no percent similarity is given for the Jena climate time series because the MPI implementation ran for several days, but never finished. This is likely due to the size of the time series and the less efficient STAMP algorithm.

Figure 6 shows an example side-by-side difference comparison between two output matrix profiles. The left side is the STUMPY Python library output. The right side is the MPI C++ Matrix Profile output. This figure illustrates how similar most of the matrix profile output was between the STUMPY implementation and our MPI implementation up to many decimal places.

IX. Performance Comparison

A simple performance comparison test was run to compare the execution time, CPU utilization, and memory usage between the reference STUMPY Python matrix profile implementation and our MPI C++ implementation. These performance tests were run on a Intel Core i5-5250

Fig. 4. Test Dataset Input Time Series Data

Filename	Datapoints	Description
AAPL.csv	254	Apple stock daily adjusted close price from 2021-02-11 to 2022-02-11
AMZN.csv	6228	Amazon stock daily adjusted close price from 1997-05-15 to 2022-02-10
AirPassengers.csv	144	Monthly airline passengers from 1949 to 1960
california_covid19_cases.csv	740	Daily California COVID-19 cases from 2020-02-01 to 2022-02-09
daily_min_temperature.csv	3649	Daily minimum temperatures from 1981-01-01 to 1990-12-31
jena_climate_2009_2016.csv	420551	Jena, Germany temperatures from 2009-01-01 to 2017-01-01
MSFT.csv	254	Microsoft stock daily adjusted close price from 2021-02-11 to 2022-02-11
TSLA.csv	253	Tesla stock daily adjusted close price from 2021-02-11 to 2022-02-10
us_gdp.csv	300	US quarterly GDP from 1947-01-01 to 2021-10-01

Fig. 5. Test Dataset Output Matrix Profile Percent Similarity

Filename	Percent Similarity
AAPL_matrix_profile.csv	99.99
AMZN_matrix_profile.csv	99.99
AirPassengers_matrix_profile.csv	100.00
california_covid19_cases_matrix_profile.csv	99.99
daily_min_temperature_matrix_profile.csv	99.99
jena_climate_2009_2016_matrix_profile.csv	N/A
MSFT_matrix_profile.csv	99.99
TSLA_matrix_profile.csv	99.60
us_gdp_matrix_profile.csv	99.99
Overall Percent Difference	99.95

processor with four cores and 16 GB RAM running Debian Linux with version 5.15.23 of the Linux kernel. Figure 7 displays the execution time in seconds needed for the matrix profile calculation to complete. Figure 8 shows the percent CPU utilization that the matrix profile calculation consumed. Values exceeding 100% indicate multiple CPU cores were involved. The maximum possible CPU utilization is 400% because the processor has four cores. Figure 9 lists the maximum memory usage in kilobytes that the matrix profile calculation used.

Note that this comparison is not fair for two reasons: 1) the STUMPY library uses a just-in-time optimized version of the STOMP matrix profile algorithm[6] whereas our MPI C++ implementation uses the original, less optimized STAMP matrix profile algorithm[1], and 2) comparing a serial Python program against a parallelized C++ program is unfair due to the massive difference in language runtimes. The differences are apparent in the performance comparison results.

The MPI matrix profile implementation is easily faster than the STUMPY matrix profile implementation for time series with a smaller number of datapoints, but is much slower for larger time series. This is due to the more efficient STOMP algorithm that STUMPY uses compared to the MPI implementation's STAMP algorithm. Note that the Jena climate time series is incomplete because our MPI implementation ran for several days but did not finish the computation.

For CPU Utilization, the STUMPY implementation only uses one CPU core for most of the time series compared to the MPI implementation using all available CPU cores. However, STUMPY does use multiple cores for the large Jena climate time series. This is due to STUMPY using the Numba just-in-time (JIT) compiler Python library that translates a subset of Python and NumPy code into native machine code. Numba supports automatic conversion of array expressions into parallel code[8], thus the higher CPU utilization for the larger Jena climate time series.

The STUMPY matrix profile implementation uses about fifteen times as much memory as our MPI implementation on average. This is expected given that STUMPY runs in a heavier garbage-collected Python runtime and with a Numba JIT compiler compared to the far more minimal C++ environment and MPI C library.

Figure 10 Shows the how many seconds it took the Python and C++ code to complete time series.

Figure 11 Shows the CPU usage percentage. 100% means 1 core, 300% means 3 cores were used.

Figure 12 Shows the amount of kilobytes that the Python and C++ programs used.

X. Conclusion

!!! WRITE CONCLUSION HERE !!!

XI. Future Work

!!! WRITE FUTURE WORK SECTION HERE !!!

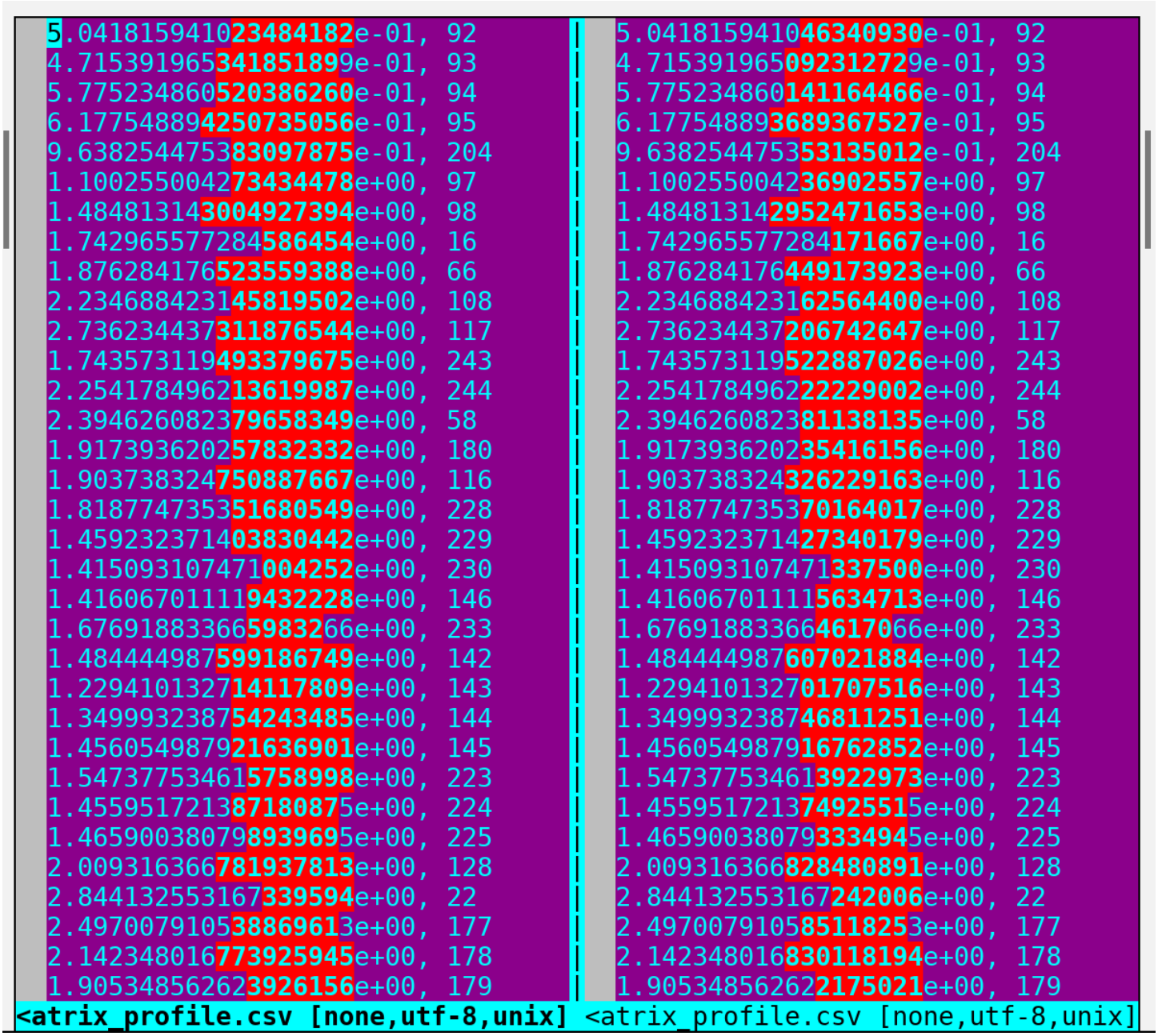

Fig. 6. Side-by-side difference comparison between matrix profile output. Left: STUMPY Python output, Right: MPI C++ output

REFERENCES

[1] C. M. Yeh et al., "Matrix Profile I: All Pairs Similarity Joins for Time Series: A Unifying View That Includes Motifs, Discords and Shapelets," 2016 IEEE 16th International Conference on Data Mining (ICDM), 2016, pp. 1317-1322, doi: 10.1109/ICDM.2016.0179.

[2] Y. Zhu et al., "Matrix Profile II: Exploiting a Novel Algorithm and GPUs to Break the One Hundred Million Barrier for Time Series Motifs and Joins," 2016 IEEE 16th International Conference on Data Mining (ICDM), 2016, pp. 739-748, doi: 10.1109/ICDM.2016.0085.

[3] Y. Zhu et al., "Matrix Profile XI: SCRIMP++: Time Series Motif Discovery at Interactive Speeds," 2018 IEEE International Conference on Data Mining (ICDM), 2018, pp. 837-846, doi: 10.1109/ICDM.2018.00099.

[4] Z. Zimmerman et al., "Matrix Profile XIV: Scaling Time Series Motif Discovery with GPUs to Break a Quintillion Pairwise Comparisons a Day and Beyond," 2019 ACM Symposium, pp. 74-86. doi: 10.1145/3357223.3362721.

[5] T. Rakthanmanon et al., "Searching and Mining Trillions of Time Series Subsequences under Dynamic Time Warping," In Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '12). 2012. Association for Computing Machinery, New York, NY, USA, 262–270. DOI:https://doi.org/10.1145/2339530.2339576.

[6] S.M. Law, (2019). "STUMPY: A Powerful and Scalable Python Library for Time Series Data Mining," Journal of Open Source Software, 4(39), 1504. [Online]. Available: https://github.com/TDAmeritrade/stumpy.

[7] E. Keogh, *The UCR Matrix Profile Page*, 2021. [Online]. Available: https://www.cs.ucr.edu/ eamonn/MatrixProfile.html.

Fig. 7. Execution Time in Seconds

Input Filename	STUMPY Python	MPI C++
AAPL.csv	15.61	0.28
AMZN.csv	16.30	7.34
AirPassengers.csv	15.42	0.11
california_covid19_cases.csv	15.43	0.91
daily_min_temperature.csv	15.78	17.71
jena_climate_2009_2016.csv	2478.40	N/A
MSFT.csv	15.42	0.27
TSLA.csv	15.52	0.19
us_gdp.csv	15.56	0.24

Fig. 8. Percent CPU Utilization

Input Filename	STUMPY Python	MPI C++
AAPL.csv	106%	338%
AMZN.csv	108%	398%
AirPassengers.csv	106%	308%
california_covid19_cases.csv	106%	374%
daily_min_temperature.csv	107%	397%
jena_climate_2009_2016.csv	385%	N/A
MSFT.csv	106%	335%
TSLA.csv	106%	330%
us_gdp.csv	106%	331%

[8] *Numba*, 2022. [Online]. Available: https://numba.pydata.org/.

Fig. 9. Memory Usage in Kilobytes

Input Filename	STUMPY Python	MPI C++
AAPL.csv	219572	13680
AMZN.csv	222204	16276
AirPassengers.csv	219816	13284
california_covid19_cases.csv	220040	13280
daily_min_temperature.csv	221240	14548
jena_climate_2009_2016.csv	431388	N/A
MSFT.csv	220040	13580
TSLA.csv	220116	13316
us_gdp.csv	220452	13360

	AAPL	AMZN	MSFT	TSLA	AirPassenger Matrix	California Covid-19 Cases	Daily Min Temp	US GDP
Python	15.27	15.67	15.14	15.22	15.14	15.13	15.38	15.27
C++	0.09	21.7	0.08	0.07	0.4	0.01	5.19	0.08

Fig. 10. Compares how many seconds it took the Python code and C++ code to complete time series calculations

	AAPL	AMZN	MSFT	TSLA	AirPassenger Matrix	California Covid-19 Cases	Daily Min Temp	US GDP
Python	106%	108%	106%	106%	106%	106%	107%	106%
C++	338%	398%	335%	330%	308%	374%	397%	331%

Fig. 11. CPU usage percentage. 100% means 1 core, 300% means 3 cores were used.

	AAPL	AMZN	MSFT	TSLA	AirPassenger Matrix	California Covid-19 Cases	Daily Min Temp	US GDP
Python	219572	222204	220040	220116	219816	220040	221240	220542
C++	13680	16276	13580	13316	13284	13280	14578	13360

Fig. 12. The amount of kilobytes the Python and C++ programs used.