

# 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 observing trends 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++ offers organizations that use MPI a way to use the Matrix Profile.

## II. PROJECT THESIS

In this project we created a minimal MPI-based 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  $t_i$ :  $T = t_1, t_2, \dots, t_n$  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,  $T_{i,m} = t_i, t_{i+1}, \dots, t_{i+m-1}$ , 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,  $D_i = [d_{i,1}, d_{i,2}, \dots, d_{i,n-m+1}]$ , 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  $1 < k < m - 1$ . Empirically, 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 stored 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,  $P = [\min(D_1), \min(D_2), \dots, \min(D_{n-m+1})]$ , 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  $T$ .

## Trivial Match Only

	7.4	6.9	14.7	19.3	17.7	19.9	15.0	8.2	8.9
7.4		10.9	7.9	15.7	18.8	19.1	15.8	1.4	8.4
6.9	10.9		16.8	16.1	13.6	18.8	14.0	11.6	6.2
14.7	7.9	16.8		16.8	19.8	18.0	19.4	8.2	13.4
19.3	15.7	16.1	16.8		20.7	23.6	18.7	15.3	11.4
17.7	18.8	13.6	19.8	20.7		19.2	23.1	19.8	14.4
19.9	19.1	18.8	18.0	23.6	19.2		14.1	20.1	20.5
15.0	15.8	14.0	19.4	18.7	23.1	14.1		16.2	16.1
8.2	1.4	11.6	8.2	15.3	19.8	20.1	16.2		8.6
8.9	8.4	6.2	13.4	11.4	14.4	20.5	16.1	8.6	

#TrivialMatch

## Exclusion Zone

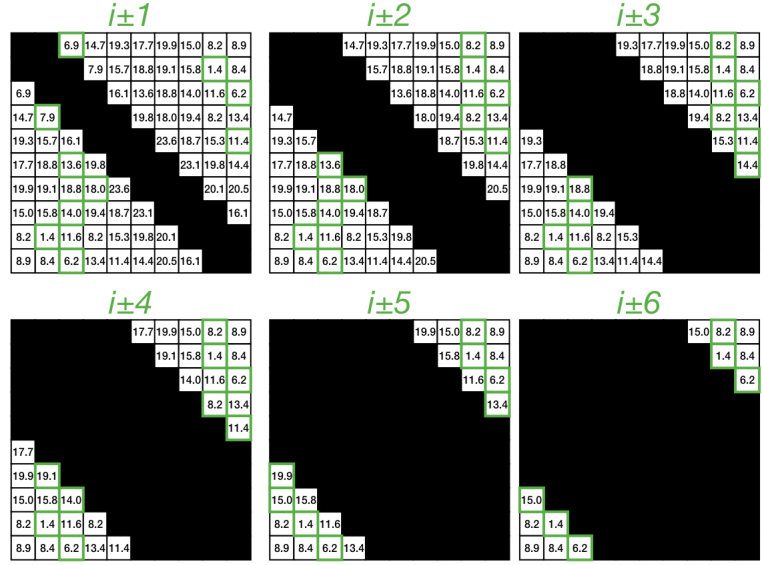


Fig. 1. Exclusion Zone Sizes

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

Fig. 2. Distance Profile Minima Collected into Matrix Profile

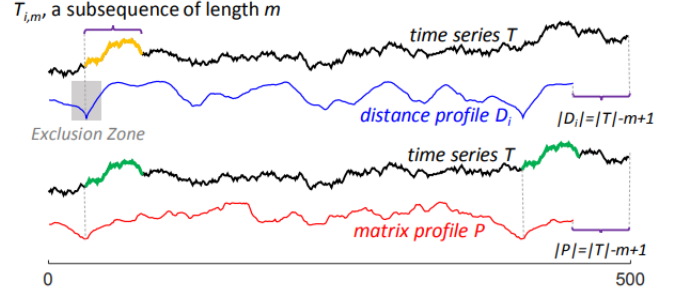


Fig. 3. Definitions Illustrated

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:  $I = [I_1, I_2, \dots, I_{n-m+1}]$ , 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, STAMPI, 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) The 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) The Leader MPI process reads in time series data from the specified input file.
- 3) The Leader MPI process `MPI_Bcasts` the time series data to all other MPI processes.
- 4) Each MPI process works on distance profile calculations for a segment of time series indices (values of  $i$ ) based on their respective 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 comparison 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 is considered correct if 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 [9] for details.

## VII. TEST DATASETS

Real world time series data was used in the test suite. Figures 4 through 12 show the input time series test datasets and the captions give a brief description of each as well as the number of datapoints in the time series. Note that many of 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.

## VIII. RESULTS

The MPI Matrix Profile implementation produced the same results as the STUMPY Matrix Profile implementation with 99.95% similarity. Figure 13 lists the percent similarity for each 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 Jena climate time series and the less efficient STAMP algorithm.

Figure 14 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 out 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 processor with four cores and 16 GB RAM running Debian Linux with version 5.15.23 of the Linux kernel.

Note that this performance 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]



Fig. 4. Apple stock daily adjusted close price from 2021-02-11 to 2022-02-11, 254 datapoints

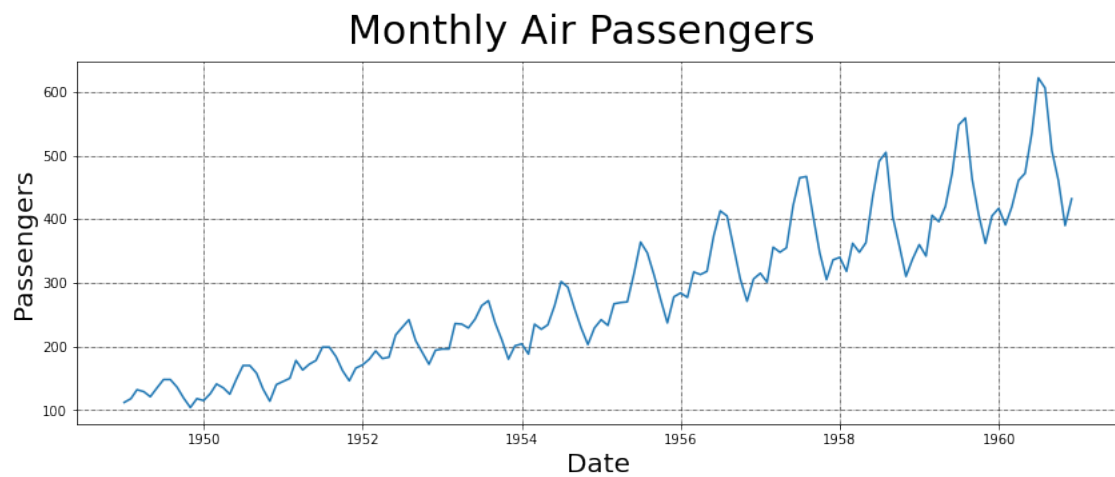


Fig. 5. Monthly airline passengers from 1949 to 1960, 144 datapoints



Fig. 6. Amazon stock daily adjusted close price from 1997-05-15 to 2022-02-10, 6228 datapoints

## California COVID-19 Cases

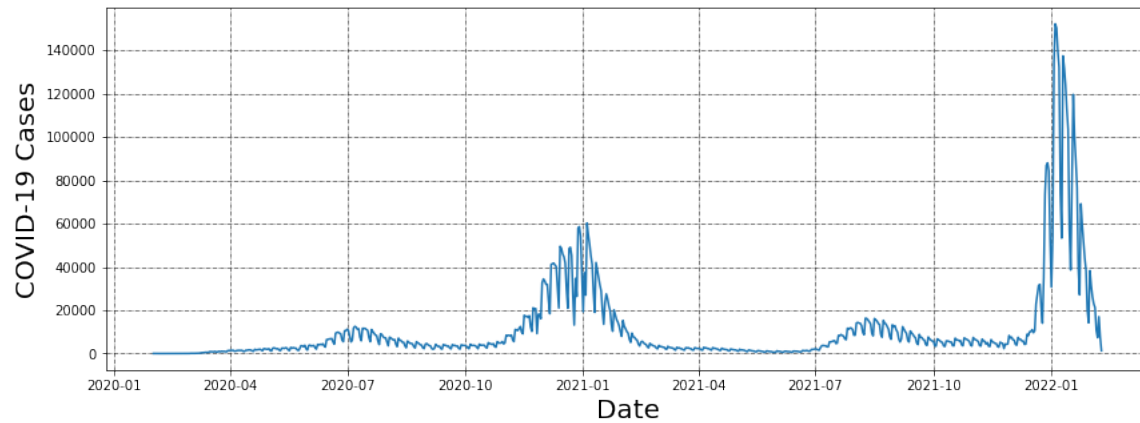


Fig. 7. Daily California COVID-19 cases from 2020-02-01 to 2022-02-09, 740 datapoints

## Daily Minimum Temperature

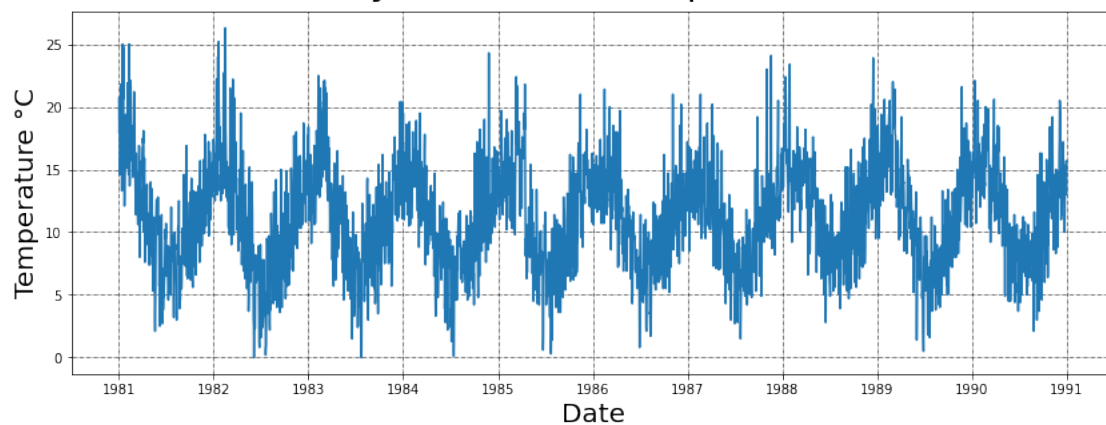


Fig. 8. Daily minimum temperatures from 1981-01-01 to 1990-12-31, 3649 datapoints

## Jena, Germany Temperatures

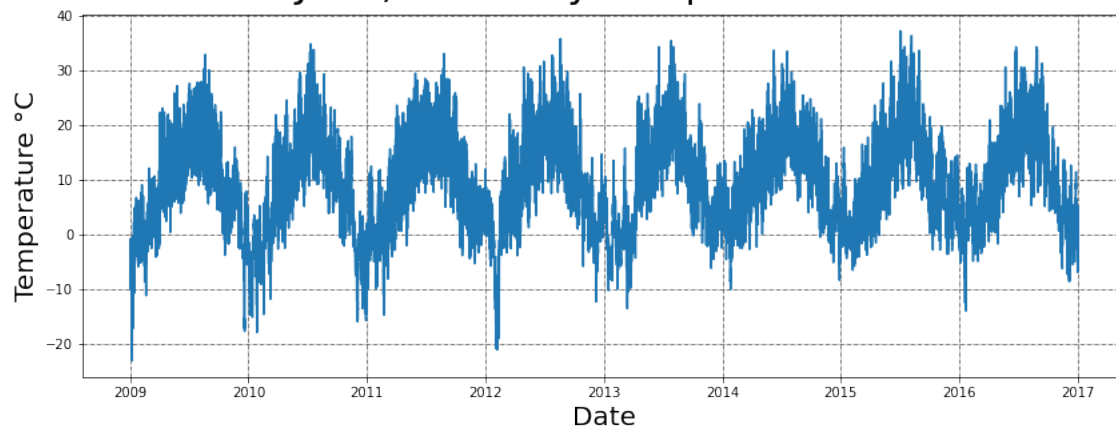


Fig. 9. Jena, Germany temperatures from 2009-01-01 to 2017-01-01, 420551 datapoints

## Microsoft Stock Price



Fig. 10. Microsoft stock daily adjusted close price from 2021-02-11 to 2022-02-11, 254 datapoints

## Tesla Stock Price

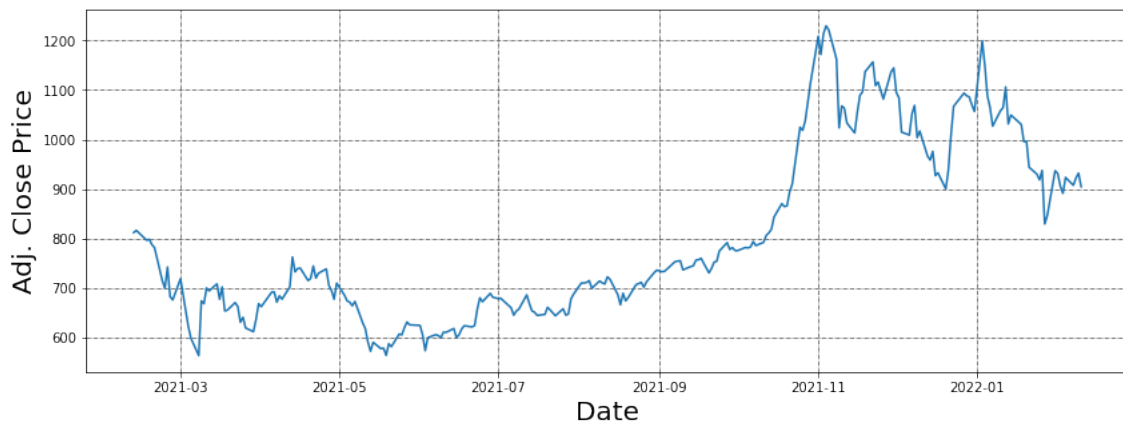


Fig. 11. Tesla stock daily adjusted close price from 2021-02-11 to 2022-02-10, 253 datapoints

## US Gross Domestic Product

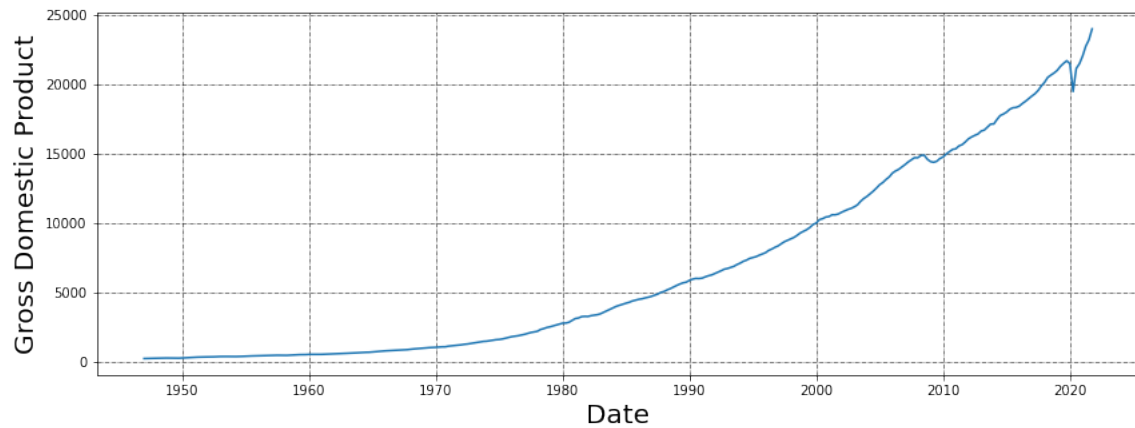


Fig. 12. US quarterly GDP from 1947-01-01 to 2021-10-01, 300 datapoints

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
TESLA_matrix_profile.csv	99.60
us_gdp_matrix_profile.csv	99.99
Overall Percent Similarity	99.95

Fig. 13. Test Dataset Output Matrix Profile Percent Similarity

5.041815941023484182e-01, 92	5.041815941046340930e-01, 92
4.715391965341851899e-01, 93	4.715391965092312729e-01, 93
5.775234860520386260e-01, 94	5.775234860141164466e-01, 94
6.177548894250735056e-01, 95	6.177548893689367527e-01, 95
9.638254475383097875e-01, 204	9.638254475353135012e-01, 204
1.100255004273434478e+00, 97	1.100255004236902557e+00, 97
1.484813143004927394e+00, 98	1.484813142952471653e+00, 98
1.742965577284586454e+00, 16	1.742965577284171667e+00, 16
1.876284176523559388e+00, 66	1.876284176449173923e+00, 66
2.234688423145819502e+00, 108	2.234688423162564400e+00, 108
2.736234437311876544e+00, 117	2.736234437206742647e+00, 117
1.743573119493379675e+00, 243	1.743573119522887026e+00, 243
2.254178496213619987e+00, 244	2.25417849622229002e+00, 244
2.394626082379658349e+00, 58	2.394626082381138135e+00, 58
1.917393620257832332e+00, 180	1.917393620235416156e+00, 180
1.903738324750887667e+00, 116	1.903738324326229163e+00, 116
1.818774735351680549e+00, 228	1.818774735370164017e+00, 228
1.459232371403830442e+00, 229	1.459232371427340179e+00, 229
1.415093107471004252e+00, 230	1.415093107471337500e+00, 230
1.416067011119432228e+00, 146	1.416067011115634713e+00, 146
1.676918833665983266e+00, 233	1.676918833664617066e+00, 233
1.484444987599186749e+00, 142	1.484444987607021884e+00, 142
1.229410132714117809e+00, 143	1.229410132701707516e+00, 143
1.349993238754243485e+00, 144	1.349993238746811251e+00, 144
1.456054987921636901e+00, 145	1.456054987916762852e+00, 145
1.547377534615758998e+00, 223	1.547377534613922973e+00, 223
1.455951721387180875e+00, 224	1.455951721374925515e+00, 224
1.465900380798939695e+00, 225	1.465900380793334945e+00, 225
2.009316366781937813e+00, 128	2.009316366828480891e+00, 128
2.844132553167339594e+00, 22	2.844132553167242006e+00, 22
2.497007910538869613e+00, 177	2.497007910585118253e+00, 177
2.142348016773925945e+00, 178	2.142348016830118194e+00, 178
1.905348562623926156e+00, 179	1.905348562622175021e+00, 179
<atrix_profile.csv [none,utf-8,unix]	<atrix_profile.csv [none,utf-8,unix]

Fig. 14. Side-by-side difference comparison between matrix profile output. Left: STUMPY Python output, Right: MPI C++ output



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 not shown because our MPI implementation ran for several days but did not finish the computation. Figure 15 displays how many seconds it took the Python and C++ code to complete the matrix profile calculations. 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 did use multiple cores for the large Jena climate time series with 385% CPU utilization (not shown on the graph). 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. Figure 16 gives the CPU usage percentage. 100% means 1 core, 300% means 3 cores were used. 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 17 shows the amount of memory in kilobytes that the Python and C++ programs used.

## X. CONCLUSION

In this project we created a minimal MPI-based implementation of the Matrix Profile in C++. It works very well for time series with a small number of datapoints, but does not scale up well to larger time series. This is primarily due to our use of the original STAMP algorithm which is much slower than the STOMP and SCRIMP++ algorithms. Figure 18 gives a graphical comparison of the STAMP algorithm's performance against the STOMP and SCRIMP++ algorithms. While the STOMP and SCRIMP++ algorithms are much more performant compared to STAMP, they are also much more complex to implement which is why we chose to implement STAMP for this project.

## XI. FUTURE WORK

The Matrix Profile is an exciting area of research for time series data mining. Most of the work is focused on mining massive time series datasets as fast as possible in cloud computing environments. Considering how versatile the

Matrix Profile is and how ubiquitous multicore processors are, there is likely a need for a fast, resource-frugal implementation of the Matrix Profile for use in portable and embedded devices such as on-board vehicle computers, field diagnostic medical equipment, smartphones, and industrial control computers.

An MPI-based implementation similar to (or perhaps derived from) our implementation could be created to use a more optimal Matrix Profile algorithm such as SCRIMP++ and outfitted with a focused domain-specific motif search set in order to quickly search an incoming data stream (a time series) for matching subsequences of interest.

For example, an on-board vehicle computer could use sensor data to rapidly recognize a hazardous road condition and alert the driver. Field diagnostic medical devices could aid doctors or first responders in quickly finding life-threatening diseases so that the proper care can be given with minimal delay. Smartphones could gain even more capabilities with on-device sensors that identify emergencies and request help without user input (e.g. active shooter situations, explosions, car accidents, et cetera). Industrial control computers could better respond to their environment by sensing failure conditions and shutting down production to prevent injuries and loss of expensive equipment.

There are many possible applications for a fast, resource-frugal, parallel implementation of the Matrix Profile. This project showed that a minimal MPI Matrix Profile implementation is possible and can be made practical with more effort.

## 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>.
- [8] Numba, 2022. [Online]. Available: <https://numba.pydata.org/>.
- [9] B. Larsen and R. McNew, *Parallel Group Project*, 2022. [Online]. Available: [https://github.com/rmcnew/parallel\\_group\\_project/actions](https://github.com/rmcnew/parallel_group_project/actions).



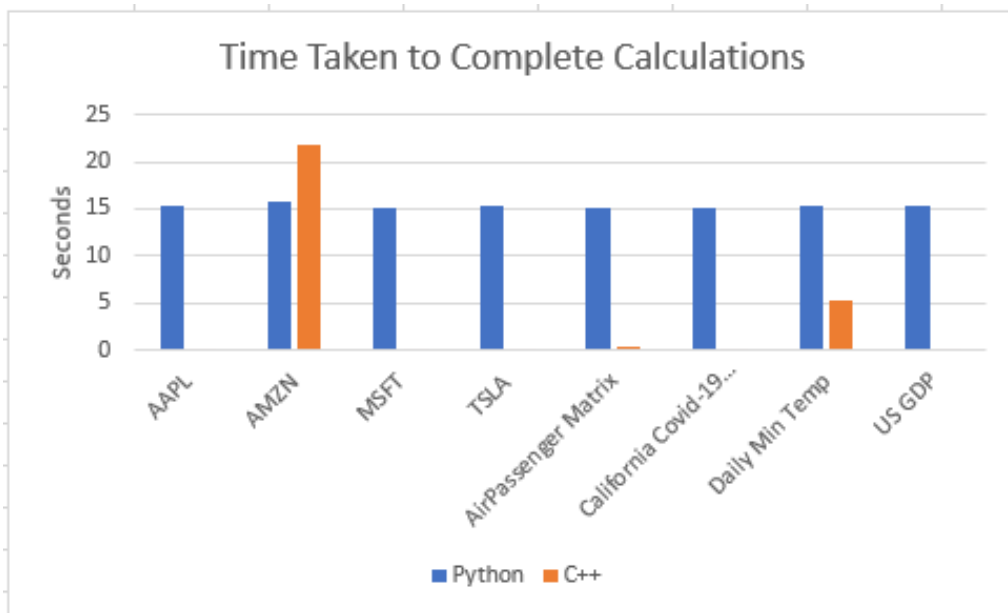


Fig. 15. Compares how many seconds it took the Python code and C++ code to complete the matrix profile calculations

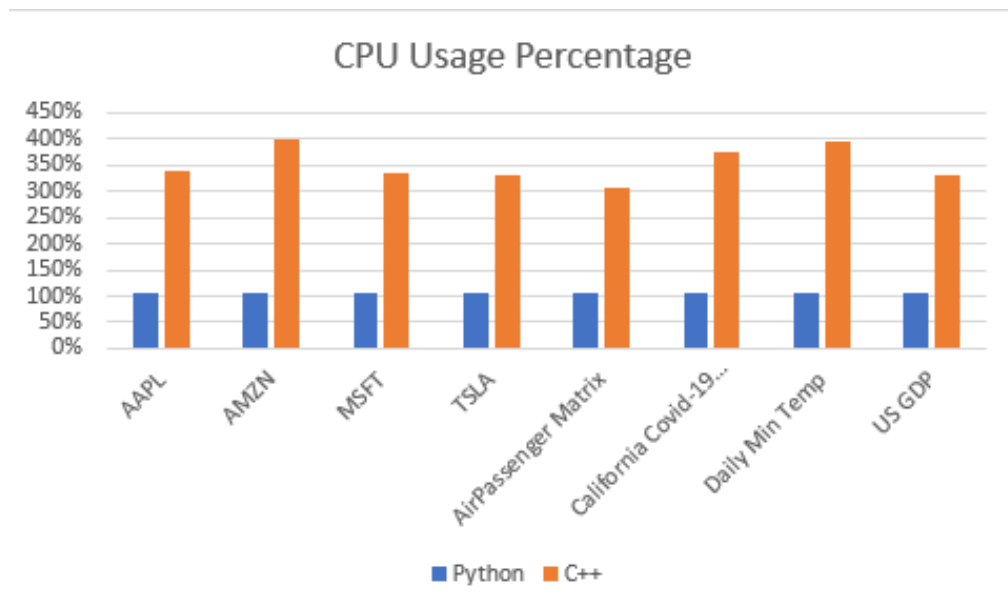


Fig. 16. CPU usage percentage. 100% means 1 core, 300% means 3 cores were used.

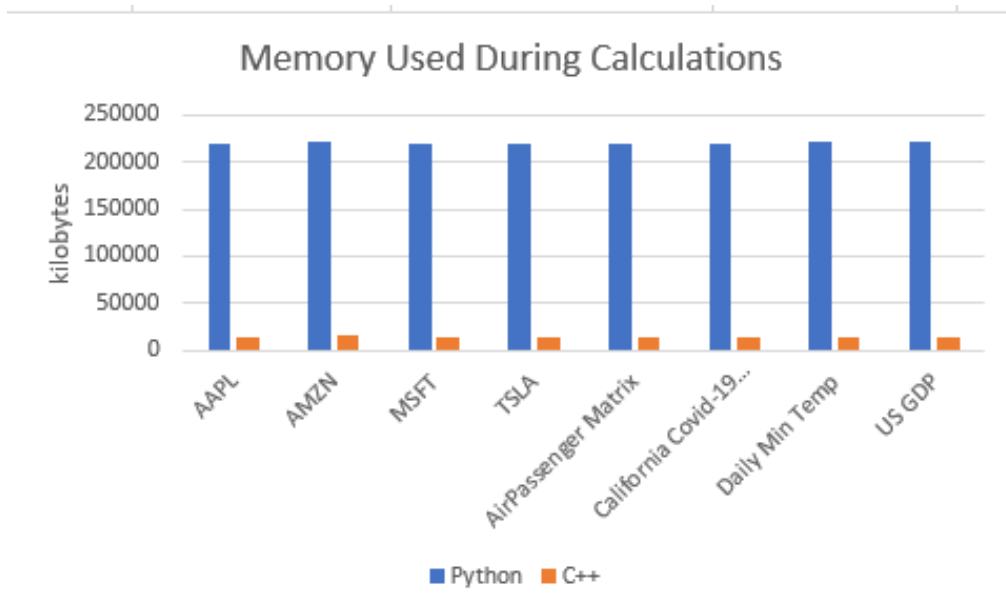


Fig. 17. The amount of memory in kilobytes the Python and C++ programs used.

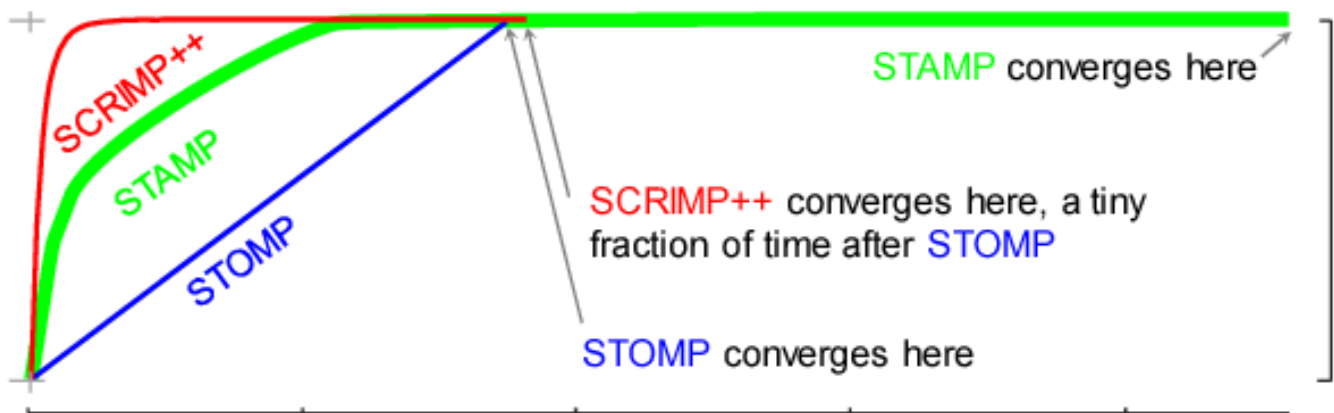


Fig. 18. Matrix Profile algorithms convergence times compared