

# Time Series Mining on High Performance Computing Systems

Amir Raoofy

Vollständiger Abdruck der von der TUM School of Computation Information and Technology der Technischen Universität München zur Erlangung des akademischen Grades eines

**Doktors der Naturwissenschaften (Dr. rer. nat.)**

genehmigten Dissertation.

**Vorsitz:**

Prof. Dr. Andreas Herkersdorf

**Prüfer der Dissertation:**

1. Prof. Dr. Martin Schulz
2. Prof. Dr. Martin Schreiber,  
Université Grenoble Alpes

Die Dissertation wurde am 21.12.2023 bei der Technischen Universität München eingereicht und durch die TUM School of Computation Information and Technology am 03.05.2024 angenommen.



# Abstract

Time series and their analysis are mainstream in many areas, from infrastructure monitoring (power grid, renewable energy generation, ...) to mobility data (self-driving cars, plane safety systems, ...), from environmental sensors (weather monitoring, data-driven architecture, ...) to factory automation (predictive maintenance, intelligent material flow, ...). As the size of such datasets grows, in many such use cases, billions of data samples and hundreds of sensors are already not uncommon, we need scalable, high-performance approaches to process them.

The mining of such datasets is crucial for modern information retrieval, analytics, and monitoring infrastructures. In particular, identifying similarities within and across large, multi-dimensional time series poses computational challenges and opportunities. Matrix profile [YZU<sup>+</sup>16] is a well-established indexing approach for the explorative analysis of time series data and serve as a promising technique for similarity indexing, creating a generic structure that encodes correlations among records and dimensions. This approach was introduced in 2016 by Yeh et al. in a series of papers [YZU<sup>+</sup>16, ZZS<sup>+</sup>18, YHK16] and has been successfully applied to mine similarities and patterns in datasets from various fields, such as seismology [SSFZ<sup>+</sup>18] and medical science [DK17], as well as used for various data mining and machine learning tasks, such as semantic segmentation, clustering, and anomaly detection [ZGS<sup>+</sup>20]. Since its introduction, due to its various performance, scalability, accuracy, and practicality advantages, it has gained significant momentum in the research community as a fundamental approach for time series mining. However, generating a matrix profile can be computationally intensive. At the same time, the modern hardware resources in HPC and the capability to scale up the resources in HPC systems can help to cope with these intensive computational costs and, therefore, significantly boost the performance, hence the throughput of explorative data science built on it.

However, the existing approaches have not widely undergone investigation on modern HPC hardware, and no extreme scalability tests, especially under High-Performance Computing (HPC) systems, have been conducted. Therefore, leveraging modern HPC hardware and petascale HPC systems remain challenging, especially for large and multi-dimensional time series.

In response to these challenges, we introduce a set of novel approaches targeting different HPC platforms—CPU-based HPC systems and GPUs—to address the efficient calculation of matrix profiles for multi-dimensional and single-dimensional time series. These approaches range from schemes for cluster-level multi-node computation and optimizations, multi-GPU computation and kernel design, reduced- and mixed-precision computation schemes, as well as algorithmic alternative methods leveraging tree data structures and schemes to ensure scalability on HPC systems.

## *Abstract*

The CPU-based approach demonstrates a sustained performance of 1.3 petaflop/s on the SuperMUC-NG system and effectively handles 128-dimensional time series datasets containing one billion records. Meanwhile, the GPU-based approach leverages reduced- and mixed-precision floating point arithmetic modes for computation and achieves significant performance improvement over the optimized single-node CPU-based approach while maintaining sufficient accuracy. This approach also scales on nodes with multiple GPUs very efficiently. The tree-based approach also shows a huge potential for matrix profile computation on HPC systems, and with the proposed optimization methods, can scale up to 1,000 nodes on the SuperMUC-NG system.

Overall, these advancements not only confirm the feasibility and scalability of the matrix profile approach for single and multi-dimensional time series mining but also pave the way for its application in large-scale, real-world scenarios. We provide comprehensive evaluations, demonstrate synthetic and real-world case studies, and discuss trade-offs between accuracy and performance, offering a multi-faceted perspective on time series analysis on HPC systems.



# Zusammenfassung

Zeitreihen und ihre Analyse sind in vielen Bereichen gängig, von der Infrastrukturüberwachung (Stromnetz, erneuerbare Energieerzeugung, ...) bis zu Mobilitätsdaten (selbstfahrende Autos, Flugzeugsicherheitssysteme, ...), von Umweltsensoren (Wetterüberwachung, Data-driven Architektur, ...) bis zur Fabrikautomatisierung (vorausschauende Wartung, intelligenter Materialfluss, ...). Da die Größe solcher Datensätze zunimmt, in vielen Anwendungsfällen sind Milliarden von Daten und hunderte Sensoren bereits keine Seltenheit, benötigen wir skalierbare, leistungsstarke Ansätze, um sie zu verarbeiten.

Das Mining solcher Datensätze ist für moderne Information Retrieval-, Analyse- und Überwachungsinfrastrukturen von entscheidender Bedeutung. Insbesondere die Identifizierung von Ähnlichkeiten innerhalb und zwischen großen, mehrdimensionalen Zeitreihen stellt eine rechnerische Herausforderung dar und eröffnet neue Möglichkeiten. Matrixprofil [YZU<sup>+</sup>16] ist ein etablierter Indexierungsansatz für die explorative Analyse von Zeitreihendaten und dient als vielversprechende Technik für die Ähnlichkeitsindizierung, die eine generische Struktur schafft und Korrelationen zwischen Datensätzen und Dimensionen kodiert. Dieser Ansatz wurde im Jahr 2016 von Yeh et al. in einer Reihe von Veröffentlichungen vorgestellt [YZU<sup>+</sup>16, ZZS<sup>+</sup>18, YHK16] und wurde erfolgreich angewandt, um Ähnlichkeiten und Muster in Datensätzen aus verschiedenen Bereichen, wie der Seismologie [SSFZ<sup>+</sup>18] und Medizin [DK17], und für verschiedene Data-Mining- und Machine-Learning-Aufgaben wie semantische Segmentierung, Clustering und Anomalieerkennung [ZGS<sup>+</sup>20] eingesetzt. Dieser Ansatz hat verschiedenen Vorteile in Bezug auf Leistung, Skalierbarkeit, Genauigkeit und praktische Anwendbarkeit. Deshalb wurde er in der Forschungsgemeinschaft als grundlegender Ansatz für die Auswertung von Zeitreihen einen erheblichen Aufschwung erfahren.

Die Erstellung eines Matrixprofils kann jedoch sehr rechenintensiv sein. Gleichzeitig können die modernen Hardwareressourcen im HPC sowie die Möglichkeit, die Ressourcen in HPC-Systemen zu skalieren, dazu beitragen, diese rechenintensiven Kosten zu bewältigen und dadurch die Leistung und damit den Durchsatz der explorativen Datenwissenschaft erheblich zu steigern.

Die bestehenden Ansätze wurden jedoch nicht umfassend auf moderner HPC-Hardware untersucht. Es wurden keine extremen Skalierungsteste, insbesondere unter High-Performance Computing HPC-Systemen, durchgeführt. Daher bleibt die Nutzung moderner HPC-Hardware und HPC-Systeme im Petascale eine Herausforderung, insbesondere für große und mehrdimensionale Zeitreihen.

Als Antwort auf diese Herausforderungen stellen wir eine Reihe neuartiger Ansätze vor, die auf verschiedene HPC-Plattformen – CPU-basierte HPC-Systeme und GPUs – abzielen, um die effiziente Berechnung von Matrixprofilen für mehrdimensionale sowie eindimensionale Zeitreihen zu ermöglichen. Diese Ansätze reichen von Methoden für

## Zusammenfassung

Berechnungen auf Clusterebene mit mehreren Knoten und Optimierungen, Berechnungen mit mehreren **GPU**s und Kernel-Design, Schemata mit reduzierter Genauigkeit sowie alternative algorithmische Methoden, die Baumdatenstrukturen und Schemata zur Gewährleistung der Skalierbarkeit auf **HPC**-Systemen nutzen.

Der **CPU**-basierter Ansatz zeigt eine anhaltende Leistung von 1,3 **petaflop/s** auf dem **SuperMUC-NG**-System und verarbeitet effektiv 128-dimensionale Zeitreihendaten mit einer Milliarde Datensätzen. Der **GPU**-basierte Ansatz nutzt reduzierter- und gemichte-Präzision Fließkomma-Arithmetik-Modi für die Berechnung und erzielt eine erhebliche Leistungssteigerung gegenüber dem optimierten **CPU**-basierten Ansatz mit nur einem Knoten, wobei eine ausreichende Genauigkeit erhalten bleibt. Der Ansatz skaliert auf **Multi-GPU**-Knoten sehr effizient. Der Baum-basierter Ansatz zeigt auch ein großes Potenzial für die Berechnung von Matrixprofilen auf **HPC**-Systemen und kann mit den vorgeschlagenen Optimierungsmethoden bis zu 1,000 Knoten auf dem **SuperMUC-NG**-System skalieren.

Insgesamt bestätigen diese Fortschritte nicht nur die Machbarkeit und Skalierbarkeit des Matrixprofil-Ansatzes für das Mining von ein- und mehrdimensionalen Zeitreihen, sondern ebnet auch den Weg für seine Anwendung in groß angelegten, realen Szenarien. Wir bieten umfassende Auswertungen, synthetische und reale Fallstudien und diskutieren Kompromisse zwischen Genauigkeit und Leistung, um eine facettenreiche Perspektive auf die Zeitreihenanalyse auf **HPC**-Systemen zu bieten.

# Preface



# Acknowledgments

I can only express my wholehearted thanks to all my teachers, professors, colleagues, students, family, and friends. Completing this work was only possible in the presence of their support.

I express my sincere gratitude to Prof. Dr. Martin Schulz, my inspiring and encouraging Ph.D. supervisor – I feel extremely lucky to have been trained by such a respectful computer scientist. I thank him for his professional approach, wise advice, his flexibility, and his dedication to my training. Also, I thank him for his invaluable feedback on my ideas and works and for many fruitful discussions throughout these years. I will never forget his passion for scientific and technical discussion on the whiteboard in his office and the many discussion sessions starting with him opening with the phrase “let’s get to the fun part”. The same goes for his midnight edits and commits to my paper manuscripts. Many thanks to my mentor Professor Carsten Trinitis who always supported me, both technically and mentally, and pushed me towards success in my Ph.D. project. I also thank Prof. Martin Schreiber for his role as examiner of this work. Also, I thank him for supporting my ideas and sharing his fun attitude during his stay at CAPS. And many thanks to PD. Dr. Josef Weidendorfer for his always-insightful technical discussions and for consistently supporting my work.

Thanks to Roman Karlstetter, with whom I worked closely for four years on lots of ideas and technical issues that “we discussed” frequently. Also many thanks for his valuable contributions to my research and publications. Additionally, I thank him for taking care of the heavy-lifting in the management and administration of Turbo and SenseE projects. My greatest thanks to Dr. Arts Yang, my former colleague who helped me greatly in onboarding to CAPS. I certainly enjoyed having many technical discussions with her, and I would like to thank her for her contributions to the development of my work. Also, thanks to my former colleague Dr. Alexis Engelke, especially for his encouraging words to consistently push me towards finishing this thesis.

Thanks to the former secretary of CAPS, Beate Hinterwimmer, RIP. I will never forget her kindness, especially the times she dropped me home on her way driving back home. Also, thanks to the new secretary of CAPS, Lisa Francke, for handling the complicated administrative issues related to my work patiently and as swiftly as possible. Also, many thanks to Jürgen Obermeier for his support and help, including in realizing prototypes of the SenseE project in the CAPS Cloud.

Thanks to all the CAPS members and colleagues for creating a professional, welcoming, supportive, and friendly environment for research, teaching, and incubating and implementing ideas. And thanks to all the students who worked together with me and hence contributed to realizing my ideas and experiments for this thesis.

## *Acknowledgments*

Thanks to [LRZ](#) for supporting this work with the compute budget, technical support and consultations, and special allocations for running system-wide jobs. Also, my colleagues at the Future Computing and Energy Efficiency group of [LRZ](#), Michael Ott and Matt Tovey, who supported and encouraged me to finish this work. I also gratefully acknowledge the Gauss Centre for Supercomputing e.V. for funding this project by providing computing time on the [GCS](#) Supercomputer SuperMUC at the Leibniz Supercomputing Centre. Also, thanks to the funding agencies [BFS](#), [ESA](#), and [EU-JU](#), which either directly or indirectly supported this work and me during the time working on this thesis.

Special thanks to my amazing partner, Simin, who lived through this journey with me. She never stopped supporting me and shared the ups and downs and the challenges. Finally, thanks to my supportive father, loving mother, and my wonderful sister, who always supported me and my education.

Amir Raoofy

December 21, 2023

# Table of Contents

<b>Abstract</b>	<b>iii</b>
<b>Zusammenfassung</b>	<b>v</b>
<b>Preface</b>	<b>ix</b>
<b>Acknowledgments</b>	<b>ix</b>
<b>Table of Contents</b>	<b>xi</b>
<b>List of Figures</b>	<b>xvi</b>
<b>List of Tables</b>	<b>xx</b>
<b>I Setting the Stage: Introduction to the Problem Context</b>	<b>1</b>
<b>1 Introduction</b>	<b>3</b>
1.1 Time Series Data and their Mining . . . . .	3
1.2 Parallel Computer Architectures and High Performance Computing . . . .	5
1.3 Motivation . . . . .	6
1.4 Problem Statement and Research Questions . . . . .	7
1.5 Contributions . . . . .	8
1.6 Structure of this Thesis . . . . .	11
1.7 Summary . . . . .	12
<b>2 Background and Technical Foundations</b>	<b>13</b>
2.1 Time Series . . . . .	13
2.1.1 Time Series Definition . . . . .	13
2.1.2 Time Series with Uniform Timestamps . . . . .	13
2.1.3 Multi-Dimensional Time Series . . . . .	14
2.1.4 Temporal and Spatial (Dimensional) Structures in Time Series . .	15
2.1.5 Time Series Windows . . . . .	15
2.1.6 Time Series Data Types and Formats . . . . .	16
2.1.7 Time Series Storage Layouts . . . . .	17
2.1.8 Representative Examples . . . . .	17

## TABLE OF CONTENTS

2.1.9	Use Case 1: Operational Data from Heavy-Duty Gas Turbines . . .	18
2.1.10	Use Case 2: Operational Data Analytics in HPC Centers . . . . .	20
2.2	Time Series Analysis and Mining . . . . .	22
2.2.1	Time Series Similarity Indexing and Similarity Join . . . . .	23
2.3	Foundations of Matrix Profile . . . . .	24
2.3.1	Matrix Profile Notations and Formulations . . . . .	27
2.3.2	Matrix Profile Illustration – Operational Data from HPC centers .	30
2.3.3	Multi-Dimensional Matrix Profile – Notations and Formulations .	32
2.3.4	Background on Matrix Profile Computation and its Costs . . . . .	34
2.3.5	Iterative In-Place Matrix Profile Computation with Enhanced Dis- tance Formulation . . . . .	36
2.4	Background on Computer Architecture and High Performance Computing	41
2.4.1	Architecture of Modern and Emerging HPC Systems . . . . .	43
2.4.2	Scalability in HPC . . . . .	45
2.4.3	Performance Optimization in HPC . . . . .	46
2.4.4	Performance Optimization Tools in HPC . . . . .	47
2.4.5	Programming Models in HPC . . . . .	48
2.4.6	GPUs in HPC and Reduced- and Mixed-Precision Computation .	48
2.5	Scope of this Work . . . . .	49
2.6	Summary . . . . .	50
<b>3</b>	<b>Overview of the Work in the Literature</b>	<b>51</b>
3.1	Scalable Methods for Time Series Mining . . . . .	51
3.2	History of Matrix Profile . . . . .	51
3.3	Methods to Compute the Matrix Profile . . . . .	52
3.3.1	Streaming (Online) and Approximate Methods . . . . .	53
3.3.2	Reduced- and Mixed-Precision Approximate Methods . . . . .	53
3.3.3	Multi-Dimensional Methods . . . . .	53
3.3.4	Matrix Profile on Accelerators . . . . .	54
3.3.5	Key Implications of Reviewing the Work in Literature . . . . .	54
3.4	Summary . . . . .	54
<b>II</b>	<b>Methods and Evaluations</b>	<b>55</b>
<b>4</b>	<b>Time Series Similarity Mining at Extreme Scale on HPC Systems</b>	<b>57</b>
4.1	Motivation . . . . .	57
4.2	Research Questions . . . . .	58
4.3	Multi-Dimensional Parallel Matrix Profile: (MP) <sup>N</sup> . . . . .	58
4.3.1	Mathematical Formulations of (MP) <sup>N</sup> Approach . . . . .	59
4.4	(MP) <sup>N</sup> for CPU-based HPC Systems . . . . .	62
4.4.1	Algorithmic Design and Optimizations . . . . .	63
4.4.2	Core-Level Design . . . . .	65
4.4.3	Node-Level Design and Parallelization . . . . .	66



4.4.4	Cluster-Level Design . . . . .	67
4.5	Putting it all Together – (MP) <sup>N</sup> Implementation in <b>MPI</b> . . . . .	68
4.5.1	Phases of (MP) <sup>N</sup> . . . . .	69
4.5.2	<b>MPI</b> Communicators in (MP) <sup>N</sup> . . . . .	70
4.6	Limitations . . . . .	70
4.7	Evaluation . . . . .	71
4.7.1	Node-Level Performance Characterization . . . . .	71
4.7.2	Characterizing Scalability . . . . .	73
4.8	Summary . . . . .	78
<b>5</b>	<b>Time Series Similarity Mining on <b>GPU</b>s with Reduced- and Mixed-Precision</b>	<b>79</b>
5.1	Motivation . . . . .	79
5.2	Research Questions . . . . .	81
5.3	<b>GPU</b> -based Approach with Reduced- and Mixed-Precision . . . . .	81
5.3.1	<b>GPU</b> -based Approach: (MP) <sup>N</sup> - <b>GPU</b> . . . . .	81
5.3.2	<b>GPU</b> Approach (Single-Tile) . . . . .	82
5.3.3	Multi-Tile <b>GPU</b> Approach . . . . .	84
5.3.4	Reduced- and Mixed-Precision Modes . . . . .	86
5.4	Implementation on <b>NVIDIA GPU</b> s . . . . .	88
5.5	Practical Approach in Assessing Accuracy . . . . .	88
5.6	Evaluation . . . . .	89
5.6.1	Performance Evaluation . . . . .	90
5.6.1.1	Performance Overview . . . . .	90
5.6.1.2	Reduced- and Mixed-Precision Performance . . . . .	92
5.6.1.3	Scalability . . . . .	93
5.6.2	Accuracy Evaluation . . . . .	94
5.6.3	Numerical Accuracy . . . . .	94
5.6.4	Practical Accuracy . . . . .	95
5.6.5	Accuracy-Performance trade-off . . . . .	98
5.7	Summary . . . . .	99
<b>6</b>	<b>Other Tackling Point; Algorithmic Approach for High-Performance Similarity Mining</b>	<b>100</b>
6.1	Motivation . . . . .	100
6.2	Research Questions . . . . .	102
6.3	Taxonomy of Methods for Computing the Matrix Profiles . . . . .	103
6.3.1	Exact Methods . . . . .	103
6.3.2	Approximate Methods . . . . .	103
6.3.3	Tree-based Nearest Neighbor Methods . . . . .	104
6.3.4	Methods Targeting <b>HPC</b> Systems . . . . .	104
6.4	Benefits of Employing Approximation and Trees for Matrix Profile Computation . . . . .	105
6.4.1	Potentials of Tree-based Methods . . . . .	105
6.5	Parallel Tree-based Approach . . . . .	107

## TABLE OF CONTENTS

6.6	Scalability Challenges and Overcoming Them . . . . .	109
6.6.1	Pipelining Mechanism . . . . .	109
6.6.2	Forest of Trees on Ensembles of Resources . . . . .	112
6.7	Scaling Behavior and Limitations of Tree Approach . . . . .	114
6.8	Evaluation . . . . .	115
6.8.1	Experimental Setup . . . . .	115
6.8.2	Baseline Comparison and Benefit Margin for Tree Approach . . . . .	115
6.8.3	Performance of the Tree Approach . . . . .	117
6.9	Summary . . . . .	121
<b>7</b>	<b>Practical Use Cases and Real-World Examples</b>	<b>122</b>
7.1	Use Case 1: Patterns in Operational Data of Heavy-Duty Gas Turbines . . . . .	122
7.2	Use Case 2: Application Classification on <a href="#">HPC-ODA</a> . . . . .	124
7.3	Real World Example for the Tree Approach: Benefits of on Real-World Datasets . . . . .	126
7.4	Summary . . . . .	128
<b>III</b>	<b>Discussions, Wrap-Up and Prospects</b>	<b>129</b>
<b>8</b>	<b>Discussions and Lessons Learned</b>	<b>131</b>
8.1	Synthesis of Key Insights . . . . .	131
8.2	Collective Effect of Presented Approaches . . . . .	132
8.3	Lessons Learned . . . . .	133
<b>9</b>	<b>Outlook and Conclusion</b>	<b>135</b>
	<b>Appendices</b>	<b>139</b>
<b>A1</b>	<b>List of Publication</b>	<b>139</b>
A1.1	Works with Direct Association to this Dissertation . . . . .	139
A1.2	Other Works . . . . .	139
<b>A2</b>	<b>Complexity of Tree-based Approach, with Pipelining and Forest Optimization</b>	<b>141</b>
<b>A3</b>	<b>Billion Scale Experiment using Tree Approach</b>	<b>142</b>
<b>A4</b>	<b>Link to Prototype Codes</b>	<b>143</b>
<b>A5</b>	<b>System and Hardware Specifications</b>	<b>144</b>
A5.1	NVIDIA <a href="#">GPU</a> s . . . . .	144
A5.2	Systems and Supercomputers . . . . .	144

*TABLE OF CONTENTS*

<b>Acronyms and Abbreviations</b>	<b>152</b>
<b>Glossary</b>	<b>160</b>
<b>Bibliography</b>	<b>163</b>

# List of Figures

1.1	Moore’s law, the original graph predicted by Moore [Moo06] (left), Performance of world’s fastest supercomputers [SDSM23] (right). . . . .	5
1.2	Three contributions represented as pillars on top of computer architecture and HPC to enable high performance and efficient time series mining. . .	9
1.3	Thesis outline. The chapters and contents of each are listed. . . . .	11
2.1	Illustration for tumbling and sliding windows. Each black box represents a sensor value, and a group of $m = 4$ consecutive are highlighted to represent a window. . . . .	15
2.2	Data generation and processing concepts in project TurbO. Time series data generated from monitoring of gas turbines is stored on disk and sent to a data center for processing. . . . .	18
2.3	Frequency spectrum of a sensor monitoring combustion process. . . . .	19
2.4	Timeline for the amplitude of combustion sensor at 160 Hz. The gaps in the data (e.g., after months seven until 17) are the result of missing monitoring data, which can have various reasons, including shutdown or maintenance. . . . .	20
2.5	HPC monitoring (time series) data stored in a database for further (e.g., offline) processing. . . . .	20
2.6	A sample from the operational data in HPCODA dataset [NMA <sup>+</sup> 19] corresponding with the collection of various performance metrics from compute nodes of LRZ systems when running HPL. . . . .	21
2.7	Illustration of matrix profile and its use for semantic segmentation of a multi-dimensional time series by finding motifs through matrix profile analysis. . . . .	26
2.8	Illustration of the distance matrix, matrix profile, and matrix profile index. . . . .	30
2.9	Illustration of similarity associations between two time series when the matrix profile index is used to create the red arcs. . . . .	31
2.10	Illustration the distance matrix and naive matrix profile computation. . .	34
2.11	Illustration of STOMP method to compute the matrix profile. We highlight the relation between the distance matrix and the matrix profile and the corresponding elements updated in each iteration (highlighted in green and blue). . . . .	37
2.12	Illustration of cross-correlation of signals (left) [Dau23] and streaming formulation for moving average (right) [Nes23]. . . . .	38

LIST OF FIGURES

4.1 Design and parallelization of  $(MP)^N$  at various levels. . . . . 62

4.2 Illustration of 3D distance matrix (upper triangular part) and the iterative computation in  $(MP)^N$ . . . . . 64

4.3 Selective combinations of the loop permutations and possible data layouts for storage in memory. . . . . 65

4.4 Illustration for the partitioning scheme of the distance matrix in  $(MP)^N$ . . 66

4.5 Illustration of all phases in our approach on a distributed memory HPC system. . . . . 69

4.6 Illustration of the virtual topology of MPI processes in communicators (left-most). Each triangle with a distinct color represents a separate MPI process working on a distinct part of the distance matrix. The red boxes represent exclusive process sets in communicators. The middle-left and middle-right figures represent communicators used in *Data Distribution* and *Aggregation* phases, and the right-most figure shows the communicator used in the I/O phases. . . . . 69

4.7 Performance breakdown of  $(MP)^N$  for a problem of size 64K and dimensionality of 128. The annotations show the percentage of time spent in sorting kernels. We report average values achieved via five repetitions for each experiment. . . . . 71

4.8 Comparing performance of various sorting kernels in  $(MP)^N$ . With  $n = 4K$  samples and  $m = 512$ . Each experiment is repeated 5 times, and we show the average result. . . . . 73

4.9 Saturation of performance in  $(MP)^N$  when increasing the number of MPI processes on a single *Skylake* node with 48 physical cores and ( $n = 10K$ ,  $d = 128$ ) vs. *Icelake* node with 72 physical cores ( $n = 64K$ ,  $d = 128$ ). The structure of  $(MP)^N$  requires a squared number of processes. The bars with asterisks hatches run with hyper-threading. . . . . 74

4.10 Results of strong scaling experiments for computing the matrix profile using  $(MP)^N$ . Parallel efficiency is annotated on top of the bars. . . . . 75

4.11 Detailed breakdown of time spent in various phases of matrix profile computation in strong scaling. . . . . 75

4.12 Detailed breakdown of time spent in various phases of  $(MP)^N$  for matrix profile computation in weak scaling. Parallel efficiency is annotated on top of the bars. . . . . 76

4.13 Weak scaling of  $(MP)^N$ , illustrating the time spent in the kernels and the main time-consuming step, i.e., *Setup*. . . . . 76

5.1 An overview of single-tile matrix profile computation approach on GPU. . 82

5.2 An illustration for the multi-tile matrix profile computation approach on GPUs. . . . . 85

5.3 Overview of various precision modes for matrix profile computation provided in  $(MP)^N$ -GPU. . . . . 86

## LIST OF FIGURES

5.4	Performance of multi-tile implementation with one tile across different generations of NVIDIA GPUs in comparison to the CPU-based (MP) <sup>N</sup> . . . . .	91
5.5	Kernel execution breakdown on NVIDIA A100 GPU. . . . .	92
5.6	Execution time and efficiency of multi-tile implementations with 16 tiles on DGX-1 ( $n=2^{16}$ , $d=2^8$ ). . . . .	93
5.7	Numerical accuracy (Here only recall rate $\mathcal{R}$ ) for the single-tile configuration of (MP) <sup>N</sup> -GPU in processing synthetic datasets compared to the CPU-based implementation. . . . .	94
5.8	Practical accuracy ( $\mathcal{R}_{practical}$ ) of single-tile implementation for pattern detection. We plot the patterns with time as x-axis ( $x \in [0, m)$ ) and the normalized values as y-axis ( $y \in [-1, 1]$ ). . . . .	96
5.9	Accuracy ( $\mathcal{F}_{classification}$ ) and performance of the nearest neighbor classifier with respect to various precision modes on the HPC-ODA dataset. . . . .	97
5.10	Accuracy-Performance trade-offs of multi-tile implementations on one NVIDIA A100 GPU with increasing number of tiles. ( $n=2^{16}$ , $d=2^6$ , $m=2^6$ ). The size of the markers indicates the number of tiles $n_{tile}$ . We annotate arrows next to the data to indicate the direction of the increase in the number of tiles. . . . .	98
6.1	Overview of various methods representing their trade-off in accuracy vs. computational efficiency . . . . .	103
6.2	Tree-based approach compared to the classical SCRIMP++ method, both iteratively progressing (single core runs, $n = 1000K$ , $m = 128$ ). . . . .	106
6.3	Weak scaling of the forefront tree-based nearest-neighbor method used for matrix profile computation. . . . .	106
6.4	Pipelining the iteration phases (P1, P2, and P3) in Pseudocode 7. Blocking <i>wait</i> operations for phases are annotated with WP1, WP2, and WP3. . . . .	110
6.5	[A]: a scenario with potential performance drawbacks due to concurrent execution of P1 and P2 on the same resource. [B]: more realistic sketch of the pipeline and latencies. . . . .	111
6.6	Single parallel tree vs. forest of trees. Pink circles represent the resources, e.g., processes. Rectangular enclosures with unique colors represent the communication contexts (i.e., MPI communicators) at different levels of a parallel tree. Each parallel tree is represented by a hierarchy of boxes with a unique color. . . . .	112
6.7	Illustration for the benefit margin for the tree approach compared to SCRIMP++. Red dots (and region) represent the areas where SCRIMP++ has better performance, while the green dots (and region) represent the areas where the tree-based method is superior. . . . .	116

LIST OF FIGURES

6.8 Single-node performance of the tree approach, with (✓) and without (✗) pipelining or forest mechanism ( $n = 6K$  per core,  $m = 512$ , random walk dataset). Configurations annotated with ✗ mark are *algorithmically* not supported. . . . . 117

6.9 Breakdown for the time spent in various phases, with (✓) and without (✗) pipelining or forest mechanisms, in weak scaling on SuperMUC-NG. ( $n = 4K$  per core,  $m = 256$ , and random walk dataset). . . . . 118

6.10 Effect of pipelining and forest mechanisms on scaling overheads ( $n = 65K$  per node,  $m = 256$ , 32 iterations, and random walk dataset). . . . . 119

6.11 Weak scaling results with (✓) and without (✗) optimizations ( $n = 65K$  per node,  $m = 256$ , 32 iterations, random walk). . . . . 120

7.1 Startup patterns in heavy-duty turbine datasets. We apply min-max normalization to avoid overflow in reduced- and mixed-precision computation. 123

7.2 Recall rate for (MP)<sup>N</sup>-GPU with various precision modes for detecting startup events in the gas turbine dataset. . . . . 124

7.3 A timeline of HPC-ODA data color-coded with the classes determined from the nearest neighbor classifier. . . . . 125

A5.1 Specification of NVIDIA V100 (left), NVIDIA A100 (middle), and NVIDIA H100 (right) GPUs. . . . . 144

# List of Tables

2.1	Example of time series data from a temperature sensor. . . . .	13
2.2	Example of a multivariate (multi-dimensional) time series. . . . .	14
2.3	Streaming dot product formulation in <a href="#">STOMP</a> . . . . .	39
3.1	Overview of methods to compute the matrix profile. . . . .	52
4.1	Iterative <a href="#">STOMP</a> formulation extended for multi-dimensional matrix profiles. . . . .	60
7.1	Categories of time series pairs, and the numbers of input time series pairs in each category in the gas turbine use case. . . . .	123
7.2	List of datasets used for the experiments and their sources. . . . .	126
7.3	Problem settings associated with each dataset used in our experiments. . . . .	127
7.4	Time to 99.9% accuracy among various methods (single-core execution). . . . .	127



## **Part I**

# **Setting the Stage: Introduction to the Problem Context**



# 1 Introduction

We are living in the *Age of Data*, where data is generated at a huge rate by modern digital and cyber-physical systems, and it is stored in and processed by information systems. Many scientific research areas, e.g., health care, weather prediction studies, and astronomy, rely on growing historical datasets. Also, a vast amount of data is generated by monitoring and data collection systems, which are equipped with many *sensors*. Moreover, social media platforms and online services store large amounts of data about the interaction of users and their transactions [Pfe19].

In many cases, this vast amount of data includes valuable information about the operation of systems and services, which is often stored in *raw* form as it arrives from sensors, without incorporating further processing to extract patterns and insights. In many cases, only through various explorative analyses and applying information retrieval methods (e.g., statistical, analytical, machine learning) this raw data can be turned into insights and knowledge, and therefore, the majority of such insight and knowledge is kept hidden in such raw data. Consequently, nowadays, *Data Science*, *Data Analytics*, and *Knowledge Discovery* in data play an important role in shaping our understanding and knowledge and in driving science and technology.

One of the main reasons behind the lack of further processing steps to extract insights from data is that many systems are not yet equipped with fully automated online data analysis pipelines. In fact, the development of such automated analytics-capable pipelines, in many cases, requires the offline explorative analysis of raw data in the first place to begin with. Also, some systems equipped with automated online data analysis still only store data in raw form, enabling fallback analysis scenarios and allowing for a more detailed (non-automated) offline analysis in case online analysis fails or does not provide sufficient insights in a certain scenario. Additionally, in many research areas, the development of such automated online analysis pipelines is almost impossible, as the systems are designed to store data in raw form with the main intention of enabling researchers to conduct explorative data analysis.

Overall, the lack of such automated online analytic pipelines highlights the importance of offline and on-demand analysis. Therefore, research on the corresponding concepts for offline analytics helps to extend the information retrieval technologies and capabilities of modern computer systems.

## 1.1 Time Series Data and their Mining

Researchers and businesses are especially interested in explorative analysis to get insights in temporal data records, often referred to as *time series*. Time series are collections

## 1 Introduction

of real-valued records arranged in chronological order. Temporal information embedded in time series associated with data records helps to analyze systematic trends, intrinsic seasonal and repeating patterns, and understand their causes [Cha04]. In particular, time series include raw information about temporal trends and correlations in data sources, which can potentially be used to create historical and predictive models as well as diagnostic insights. Therefore, in modern knowledge discovery and information retrieval, time series mining plays an important role. The term mining in this context mainly refers to the extraction of patterns, e.g., frequently repeating patterns (sometimes referred to as motifs), groups of correlated records (often referred to as clusters), and unusual records (often known as anomalies) in time series data. Especially with the ever-growing presence of the Internet of Things and surveillance systems in industry, the time series mining topic is gaining a rising momentum, however, despite the recent increasing interest, extracting meaningful insights in time series is not limited to the modern information age; time series analysis has contributed to human knowledge for a few centuries: In the 17th century, even before there was a significant record-keeping and analysis infrastructure in health care and medical practices, John Graunt [Gra62] published *the first life table*, which includes a statistical analysis on top of the historical data about the age of people at the time of death in the city of London [Nie20].

While John Graunt’s life table includes only simple statistical analysis of time series data, later, in the 1920s, the first autoregressive models were applied to real-world time series data in astronomy [Nie20].

However, it was only after the dawn of the computer age that time series analysis was extended beyond such simple statistical modeling. Major progresses in time series analysis and mining appeared after the introduction of the concept of general-purpose computers as Universal Machine by Alan Turing in 1936 [Tur50], followed by the invention of the Turing-complete Z3 computer by Konrad Zuse in 1941 [DBp23], after the appearance of the Von Neumann architecture in 1945 [vN45], all of which started and drove the architecture and computational model of today’s computers. In particular, the first fundamental contribution to time series analysis appeared in 1970 in the statistics textbook *Time Series Analysis, Forecasting and Control*, where the Autoregressive Integrated Moving Average (ARIMA) models were introduced, enabling time series modeling and forecasting [BJ70].

Still today, advances in computer systems in the information age are major drivers and enablers in time series analysis and mining. In particular, exponential growth in compute power of computer systems, as predicted by *Moore’s law* (Figure 1.1 left), as well as parallel architectures and high performance computing (HPC) systems, enable more complex analyses and modeling, e.g., using classical analytic and modern machine learning methods. Moreover, the advances in data collection and database, data management, and handling systems enable the storage of large historical data for data-driven modeling and analysis as well as more structured collection and representation of time series datasets.

## 1.2 Parallel Computer Architectures and High Performance Computing

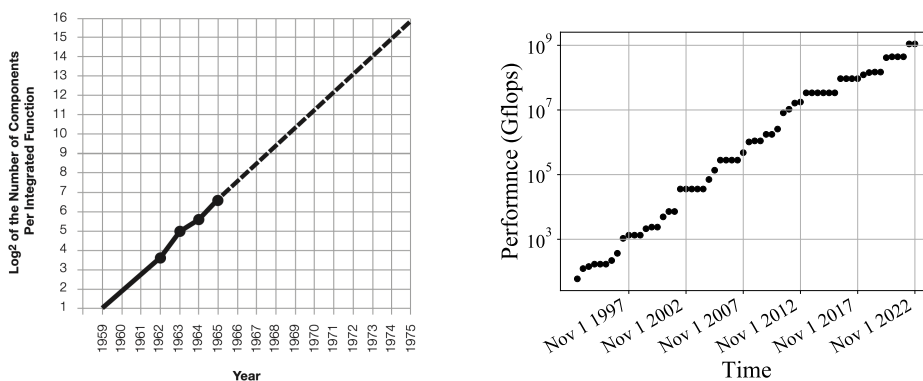
The performance of computer systems directly correlates with the technological advances in processors, memory, accelerators, networking, and storage technologies. As predicted by Gordon E. Moore, a co-founder of Intel Corporation, technological advances were expected exponential [Moo06]:

“The complexity for minimum component [per integrated function, i.e., integration circuitry] cost has increased at a rate of roughly a factor of two per year (Figure 1.1 left).”

This corresponds to the number of transistors in an integrated circuit (IC) doubling every two years, which is reflected in the performance of computer systems at all levels. This trend has remained in place for more than 40 years and is known as [Moore’s law](#). Although the growth rate has been slowed down only recently due to technological limitations and challenges, the growth trend still remains exponential.

While the concepts of parallelism in hardware and the corresponding technologies have been around much longer, especially since 2005, one major factor that has contributed to realizing the exponential growth in compute power is parallelization. This consists of parallelization both in hardware (at various architectural levels) and software. With respect to hardware, this includes instruction level and memory parallelism, vectorization and data parallelism, multi-core multi-processing, and multi-node computing. Therefore, to benefit from the computing power of modern computer systems, in particular, to push the edges of time series mining, we need to address parallelism at various levels and aspects of computer systems (e.g., in software and hardware).

Similar exponential growth in the compute power of the world’s fastest supercomputers is observed (Figure 1.1 right) over the last 29 years, as reported in the [TOP500](#) list [SDSM23]. This infamous list was established in 1993 by Hans Meuer, Erich Strohmaier, and Jack Dongarra to list the fastest 500 supercomputers around the world based on



**Figure 1.1:** [Moore’s law](#), the original graph predicted by Moore [Moo06] (left), Performance of world’s fastest supercomputers [SDSM23] (right).

## 1 Introduction

the performance of High Performance LINPCK HPL benchmark [DMBS79]. This list is updated twice a year at the two major HPC venues, the International Conference for High Performance Computing, Networking, Storage and Analysis and the International Supercomputing Conference. Recently, this exponential growth has led to exascale systems: the world’s fastest supercomputer and the first system ever to break the exaflop/s barrier, i.e., 1,000,000,000,000,000 floating point operations per second (flop/s), Frontier, was installed in US DOE Oak Ridge National Laboratory by Hewlett Packard Enterprise, which is a massively parallel computer consisting of 9,400 compute nodes equipped with AMD Instinct MI250X GPUs.

### 1.3 Motivation

Similar to many other workloads, large-scale supercomputers can be beneficial for time series analysis and mining by allowing massive parallelization through leveraging horizontal scaling of the compute resources, which is defined as adding additional resources, e.g., compute nodes. Such scaling increases the throughput for explorative data analytics (i.e., analysis and mining of time series), by leveraging lots of compute nodes, memory, and networking bandwidth, as well as storage through parallelism. An example of such analysis workloads, which is also the time series mining method targeted in this thesis, is the *matrix profile* approach. Matrix profile is a similarity indexing approach based on nearest neighbor analysis of time series windows. There exist several methods to address matrix profile computation that can benefit from extreme scaling on large-scale supercomputers. However, despite the potential, there is little research in the literature in this direction. Additionally, extreme scaling of an application on a large-scale supercomputer is not a trivial task and requires addressing various challenges in high-level and low-level programming interfaces, managing parallelisms, and overcoming scaling overheads. Often, these challenges are specific to the particular application at hand, and therefore, scaling challenges specific to time series mining applications require special consideration.

Exploiting systems with architectures and hardware features that could suit time series mining is another promising aspect. Specifically, an increasing portion of the TOP500 systems exploit Graphics Processor Units (GPU) as accelerators. For instance, seven out of the top ten systems in the latest list are GPU-based systems. Also, GPUs are proven efficient and, therefore, are widely used in the context of artificial intelligence AI and machine learning (ML), as they provide a vast number of compute cores, large memory bandwidth, and special hardware support for reduced-precision arithmetics suitable for AI and ML applications. A similar growing interest in using the power of GPUs for data analytics and mining can be observed. Therefore, exploring GPUs and their capabilities to bring efficiency and performance to time series mining is another promising research direction.

Improving the scaling efficiency of time series mining on supercomputers is another viable research direction. In fact, such improvements are required to time series mining workload to efficiently use the massive parallelism and abundant resources of HPC sys-

tems. In the context of parallel computing, the speedup is defined as the ratio of sequential to parallel elapsed time for computation. The theoretical limits for the maximum speedup and, therefore, the parallel efficiency of applications relate to the portion of the run time that does not benefit from parallelization. The theoretical limits are calculated using the two well-known Amdahl’s [Amd67] and Gustafson’s Lawson’s laws [Gus88] related to strong and weak scaling limits, respectively. In strong scaling, the number of resources (e.g., compute cores) is scaled to compute a fixed problem size, while in weak scaling, the problem size scales with the number of resources. For efficient scaling, we need to inspect the scaling bottlenecks and overheads where the mining codes cannot benefit from parallelism. In such cases, novel algorithmic mitigations can help to address these bottlenecks.

### 1.4 Problem Statement and Research Questions

The main challenges addressed in this work are around enabling and quantifying high-performance and efficient time series mining at a large scale for large time series datasets. The rationale behind scoping this work around these challenges is the need to process growingly larger time series datasets in real-world applications. Therefore, we employ various large-scale supercomputers and accelerators as well as investigate various code optimizations and HPC techniques applied to time series mining.

We focus on time series with uniform time stamps, as they prominently appear in many monitoring and research scenarios. We consider the growing size of time series datasets in real-world scenarios a central challenge and address this challenge by improving the parallel efficiency, performance, and scalability of the time series mining methods.

We consider single- and multi-dimensional time series data and the methods dealing with mining them. In particular, we focus on the offline analysis and mining of time series using the matrix profile approach as a representative workload for the offline time series mining domain. The Matrix profile approach long serves as a fundamental data mining method to investigate time series [YZU<sup>+</sup>16, Keo23, JRY<sup>+</sup>22], and provides a way to detect similar structures (patterns) when comparing two input time series. Matrix profile is widely accepted in the data science community and has been successfully applied to various application domains, including the investigation of earthquake foreshocks [SSFZ<sup>+</sup>18], analysis of power system events in synchrophasor data [SYK<sup>+</sup>19], music information retrieval (MIR) [SYBK16], similarity searching of bacteria’s DNA [YZU<sup>+</sup>18], and others.

We investigate variants of matrix profile computation approaches and highlight their efficiency, performance, and scalability. We specifically investigate the benefits of GPU architectures as accelerators, leveraging large-scale supercomputers and HPC techniques for matrix profile computation. All these investigations are aligned with the general scope of this work in extending computational capabilities to address growingly larger time series datasets in real-world applications.

We structure the research in this work around three main high-level research questions (here represented with Q) that form the foundations of the main three contributions of this work:

## 1 Introduction

- Q1: how to compute the matrix profile on large-scale **HPC** systems with high performance, e.g., at **petaflop/s** rate?
- Q2: how modern **HPC** compute nodes, that are commonly equipped with multiple **GPUs**, can be used for matrix profile computation with high efficiency, allowing to exploit hardware features like reduced-precision arithmetic capabilities of modern hardware.
- Q3: how the conventional **HPC** computation, parallelization, and optimization techniques, such as the use of efficient algorithms and data structures, communication latency hiding, and controlling the communication granularity, can be leveraged for the computation of matrix profile.

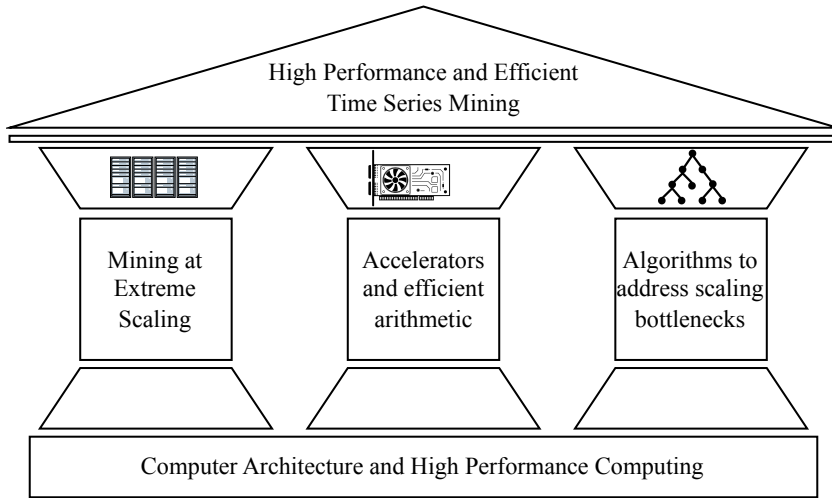
We structure this work based on new methods and approaches and conduct experimental evaluations to address these questions, covering various aspects of the application domain, hardware domain, and optimizations. For each research question, we select a specific time series mining problem, i.e., matrix profile computation for a certain type/category of time series. We then develop new concepts and implement corresponding solutions on top of a certain hardware architecture or **HPC** system and conduct evaluations on **HPC** systems. In particular, the Leibniz Supercomputing Centre (Leibniz Rechenzentrum) (**LRZ**) is an institute of the Bavarian Academy of Sciences and Humanities [dW23] (Bayerische Akademie der Wissenschaften, **BAdW**) in Garching near Munich, Germany, houses several clusters, including the *LRZ AI systems*, *LRZ BEAST systems*, and the flagship system, the *SuperMUC-NG*. The various architectures and hardware available at different systems in the **LRZ** are the core enablers for the development of this thesis. Specifically, the flagship system has significant importance as it allows for investigating and addressing time series mining challenges for larger datasets at larger scales. At the time of writing this thesis, as of November 2023 **TOP500**, it is standing at the 29th position in the **TOP500** list of the fastest supercomputers with 19.4 **petaflop/s** Linpack [DMBS79] performance.

### 1.5 Contributions

Figure 1.2 provides an overview of the key contributions of this thesis. We illustrate the contributions as three pillars on top of the computer architecture and **HPC** foundations as the key enablers to efficient and high throughput time series mining.

Contribution #1: we develop a novel approach for calculating the matrix profile for large multi-dimensional time series in parallel on **HPC** systems at large scale. We provide a scalable implementation of this method and study its performance on a **petascale** system.





**Figure 1.2:** Three contributions represented as pillars on top of computer architecture and HPC to enable high performance and efficient time series mining.

State-of-the-art algorithms for the computation of a matrix profile mostly target one-dimensional time series, i.e., a single time series covering one sensor input. Only recently, the first approach targeting multi-dimensional time series appeared in the work of Yeh et al. [YKK17] offering detailed insights into repeating patterns across different time series, significantly increasing the ability to understand multi-dimensional time series. However, these approaches are significantly more compute-intensive than one-dimensional matrix profile algorithms and, hence, are no longer feasible on standard systems for realistic workloads. Existing approaches have not been shown to scale to larger systems, nor can they be used on anything but relatively small datasets. In our work, we present the first scalable solution— $(MP)^N$ , stands for (Multi-dimensional Parallel Matrix Profile)—for the mining of large-scale multi-dimensional time series targeting CPU-based HPC systems. This approach enables the computation of large matrix profiles—as a modern data mining approach—on an HPC system. Through leveraging the HPC system, this approach is proved to be applicable to large-scale real-world problems, providing highly scalable throughput and accuracy.

Contribution #2: we introduce the first method (and its implementation) to compute the matrix profile for multi-dimensional time series on GPU(s). We discuss several reduced- and mixed-precision modes for computing multi-dimensional matrix profiles on GPUs and evaluate them based on their performance and accuracy.

While the state-of-the-art approach for computing multi-dimensional matrix profiles targets CPU-based systems [RKY<sup>+</sup>20], exploiting GPU systems for this workload is promising: previous studies show that this workload is memory-bound [Pfe19], suggest-

## 1 Introduction

ing that leveraging the High Bandwidth Memory (HBM) of GPUs promises performance improvements. Additionally, as this workload is not communication bound, the throughput is expected to scale with multiple GPUs. However, the use of GPU requires redesigning the parallelization scheme (i.e., the parallelization scheme introduced in (MP)<sup>N</sup>) and the underlying data layouts. On top of that, the problem of finding similar (and not necessarily exactly identical) patterns in multi-dimensional time series offers the door to reduced- and mixed-precision calculation schemes, which have not been investigated before. Reduced- and mixed-precision schemes, aside from improving performance, can also reduce the memory footprint, resulting in an even more efficient usage of the GPU memory bandwidth and the ability to support larger problems. However, they naturally lead to more numerical errors and, hence, new challenges to preserve acceptable numerical accuracy. In this work, we extend the state-of-the-art approach for the multi-dimensional matrix profile computation and introduce a new algorithm for GPUs that addresses the mentioned data management, kernel, and arithmetic design challenges.

Contribution #3: we develop a new approach for approximate computation of matrix profiles that is based on the iterative nearest neighbor approximation. This approach leverages tree-based data structures and targets the single-dimensional matrix profiles. We extend the state-of-the-art iterative nearest neighbor scheme with new optimization mechanisms to better scale the computation on large HPC systems.

The matrix profile is often computed using the well-known classical *exact* approach to compute the underlying nearest-neighbor problem. These approaches rely on exhaustive exact search operations to find the nearest neighbors [YZU<sup>+</sup>16, YZU<sup>+</sup>16, HHvW<sup>+</sup>20]. Due to the costs of exact search operation, these approaches are generally inefficient for *large* datasets as the computational costs for addressing these search operations scale quadratically with the size of the datasets (i.e., the number of records). Additionally, these approaches typically rely on mitigating the computational costs for the search operations by extensive arithmetic optimizations of compute kernels [ZYZ<sup>+</sup>18], or use accelerators in computation [ZZS<sup>+</sup>18, JRY<sup>+</sup>22], and deploying on a cloud-based [ZKS<sup>+</sup>19] or HPC systems [RKY<sup>+</sup>20, Pfe19]. On the other hand, *approximate* approaches [SZSF<sup>+</sup>19, ZYZ<sup>+</sup>18] (i.e., methods to compute a solution that approximates the matrix profile) are drawing increasing attention as they can provide solutions that are much more efficient to compute while also being accurate enough in practice. However, all existing approaches still suffer either from excessive computational costs [ZYZ<sup>+</sup>18], are restricted to specific settings [SZSF<sup>+</sup>19] and application scenarios [LWM<sup>+</sup>22], or lack parallelization. No research in the literature focuses on classical nearest-neighbor approaches for the computation of matrix profiles to prune the search space and reduce computational costs. In this work, we focus on these approaches, specifically on the family of the iterative approximate nearest-neighbor [Cla83] exploiting *tree* data structures and randomized approximate nearest neighbor approach [Ben75, AMN<sup>+</sup>98] in distributed memory set-

ting on HPC systems [XB16] as state of the art. We address computation for large datasets and target large-scale CPU-based HPC systems. We demonstrate that, when applied to matrix profile computation at a large scale, the state-of-the-art method suffers from excessive communication and, therefore, scaling overheads. We address these scaling overheads and, with that, introduce a new alternative approach for matrix profile computation.

## 1.6 Structure of this Thesis

We provide an overview of the structure of this thesis in Figure 1.3. This thesis is structured in three main Parts, where each part includes two to four chapters covering the main contents of this work.

The first Part includes three chapters and covers an introduction to the context of the time series mining problem, specifically matrix profile computation, in the realm of HPC. This Part includes various introductory topics, from background on time series and their mining to an overview of existing methods and approaches in the literature. Additionally, we cover basic concepts from computer architecture and HPC in this chapter.

The second part includes four chapters introducing the contributions of this work. The first three chapters introduce the main methodological contributions of this work and their evaluations. Each of these three chapters starts with a short motivational description followed by a section discussing research questions running through the chapter. After presenting the methods, the evaluations are proposed to answer the research questions. The last chapter in this part includes illustrative examples and practical use cases of the presented methods.

Parts	Chapters	Contents							
Setting the Stage: Introduction to the Problem Context	Chapter 1: Introduction	Time Series Data and their Mining	Parallel Computer Architectures and High Performance Computing	Motivation	Problem Statement and Research Questions	Contributions	Structure of this Thesis		
	Chapter 2: Background and Technical Foundations	Time Series	Time Series Analysis and Mining	Foundation of Matrix Profile	Background on Computer Architecture and High Performance Computing	Scope of this Work			
	Chapter 3: Overview of Work in the Literature	Scalable Methods for Time Series Mining		History of Matrix Profile		Methods to Compute Matrix Profile			
Methods and Evaluations	Chapter 4: Time Series Similarity Mining at Extreme Scale on HPC Systems	Motivation	Research Questions	Multi-dimensional Parallel Matrix Profile: $(MP)^N$	$(MP)^N$ for CPU-based Systems	Putting it all Together - $(MP)^N$ Implementation in MPI	Limitations	Evaluation	
	Chapter 5: Time Series Similarity Mining on GPUs with Reduced and Mixed Precision	Motivation	Research Questions	GPU-based Approach with Reduced and Mixed Precision	Implementation on NVIDIA GPUs	Practical Approach in Assessing Accuracy	Evaluation		
	Chapter 6: Algorithmic Approach for High-Performance Similarity Mining	Motivation	Research Questions	Taxonomy of Methods for Computing the Matrix Profiles	Benefits of Employing Approximation and Trees for Matrix Profile Computation	Parallel Tree-based Approach	Scalability Challenges and Overcoming Them	Scaling Behavior and Limitations of Tree Approach	Evaluation
	Chapter 7: Practical Use Cases and Real-World Examples	Case Study 1 for $(MP)^N$ -GPU: Patterns in Operational Data of Heavy-Duty Gas Turbines		Case Study 2 for $(MP)^N$ -GPU: Application Classification on HPC-ODA		Real World Example for the Tree Approach: Benefits of on Real-World Datasets			
Discussions, Wrap-Up and Prospects	Chapter 8: Discussions and Lessons Learned								
	Chapter 9: Outlook and Conclusion								

**Figure 1.3:** Thesis outline. The chapters and contents of each are listed.

## 1 Introduction

Finally, the last part includes discussion and conclusion chapters.

### 1.7 Summary

In this chapter, we provided an introduction to the context of this thesis. We highlighted the importance of time series data and their mining in modern information retrieval. We then emphasized the importance of exponential development in the capabilities of computer systems and how modern [HPC](#) systems could play an important role in advancing the mining of large time series datasets. We provided an overview of the motivations of this work and the research questions driving this thesis. We also briefly discussed an overview of the contributions of this work.

## 2 Background and Technical Foundations

### 2.1 Time Series

A time series represents measurement records of variables or sensors at regular times, for example, the temperature levels of a city. Time series data is one of the most common data types that capture activity records involving temporal measurements across a wide range of domains. Especially modern digital, cyber-physical, and surveillance systems store data in some sort of time series data and at ever-increasing huge rates.

#### 2.1.1 Time Series Definition

Time series can be defined as a collection of real-valued numbers with timestamps [JRY<sup>+</sup>22], often related to historical data records corresponding to values reported by a sensor. More formally, a tuple of `<timestamp, sensor value>` represents an individual record and the collection of these records is defined as time series. Such representation of historical data indexed by timestamps provides valuable insight into the past, trends, and evolution of records [DS21]. Table 2.1 shows an example of a time series dataset from a real-world application [KWFH<sup>+</sup>19]. The sensor values are readings of a temperature sensor (in °C) ordered according to the date when the sensor reading was done. Timestamps are typically represented as UNIX timestamps [Int], here in nanoseconds passed since January 1st, 1970, at UTC. Each row in this table corresponding to a single sensor reading is called a *sample* or *record*.

nanosecond timestamp	corresponding date	sensor value
1546435499657996160	2019-01-02 13:24:59.657996160	53.759998
1546435501405997760	2019-01-02 13:25:01.405997760	53.790005
1546435501736996160	2019-01-02 13:25:01.736996160	53.790005
1546435503814993920	2019-01-02 13:25:03.814993920	53.790005
1546435505892991680	2019-01-02 13:25:05.892991680	53.759998
1546435507217996160	2019-01-02 13:25:07.217996160	53.759998

Table 2.1: Example of time series data from a temperature sensor.

#### 2.1.2 Time Series with Uniform Timestamps

Similar to the majority of work on matrix profile and time series, in this work, we work on time series with uniform timestamps, which rely on consecutive timestamps with

## 2 Background and Technical Foundations

constant differences. In some scenarios, similar to the data presented in Table 2.1, we deal with *non-uniform* timestamps, where the time interval between two consecutive records is not constant. In such cases, the timestamp itself or its progression should be present in the dataset to allow for meaningful representation, storage, and later-on analysis. Similar to some sources, we make a distinction for time series with non-uniform timestamps and call it *temporal data* instead of time series. However, most commonly, in many use cases, the timestamps are *uniform*, meaning that the time interval between two consecutive records is constant. In such cases, the time series representation, storage, and analysis are simpler and can mainly be represented by the sensor values and sampling rate (or sampling frequency). More specifically, in this case, the representation of the time series can be reduced from the collection of tuples `<timestamp, sensor_value>` to just a vector of `<sensor_value>` annotated with the sampling rate as an additional meta information. As we are focusing on time series with uniform timestamps, from hereon, we use *time series* to refer to the case with uniform timestamps.

### 2.1.3 Multi-Dimensional Time Series

In a more general setting, a time series can consist of data records corresponding to multiple variables, often related to values reported by multiple sensors, which are represented in multiple dimensions. In this setting, the individual records can be represented as tuples of `<timestamp, val_sensor1, val_sensor2, ..., val_sensord>`, where each of `val_sensori` represent the value reported by sensor<sub>i</sub> at the particular `timestamp`. In the common case, these `d` dimensions (or variables) often correspond to sensors for monitoring/observing different aspects of the same process simultaneously, and therefore, they might be interrelated, structured, or correlated. For example, consider the sequence of temperature and pressure levels in a city, reported by the corresponding sensors, to form a two-dimensional time series. Based on the number of sensors (or variables) in the time series, we use terms *single-dimensional* (univariate) with a single sensor (or variable) and *multi-dimensional* (multivariate) time series with two or more sensors (or variables).

Table 2.1 provides an example of a single-dimensional time series, and Table 2.2 presents an example of the multi-dimensional case. More specifically, Table 2.2 provides a two-dimensional time series gathered from the real-world operation of a gas turbine [KWFH<sup>+</sup>19]. The sensor readings correspond to the speed of the turbine (`sensor1`) and the generated power by the turbine (`sensor2`) represented by 1 HZ sampling rate.

timestamp (seconds)	power (MW)	speed (RPM)
0.00	11.36	300.100006
1.00	11.39	300.100006
2.00	11.39	300.100006
3.00	11.41	300.000000
4.00	11.41	300.000000

**Table 2.2:** Example of a multivariate (multi-dimensional) time series.

The multi-dimensional time series presented in Table 2.2 has uniform timestamps, and therefore, specifying 1 HZ as the sampling rate is sufficient here and can be used as an alternative representation to omit the timestamp column in Table 2.2.

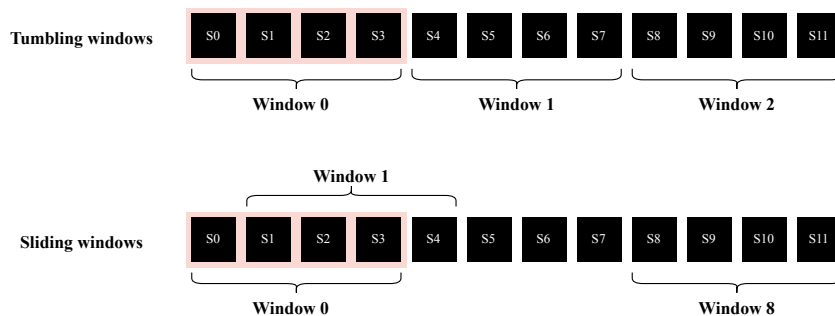
### 2.1.4 Temporal and Spatial (Dimensional) Structures in Time Series

One of the main reasons to store time series data is to keep historical records of the behavior of processes and systems and potentially later gain valuable insights from the data. This historical data can be used to analyze, model, predict, and understand the processes. Such analyses and modeling often come down to understanding structures, repetitions, and correlations within data, which we refer to as temporal and spatial structures here.

One important intrinsic feature of data, when represented in the form of time series, is the presence of temporal relations, information, and structures (features) encoded in the sequence of samples. Analyzing these temporal structures can provide valuable information about trends, seasonality, and other features in historical data. On the other hand, in the case of multidimensional time series data, analyzing the structures and correlations among various dimensions (sensors) can provide valuable insight as well, e.g., into the relations and correlations among various phenomena and sensors.

### 2.1.5 Time Series Windows

When analyzing temporal features within the time series, the time series windows play a key role as temporal feature primitives. Time series windows can be employed to represent local features, temporal patterns, and repeating structures and, therefore, are commonly used in time series analysis scenarios. These are slices of the time series that bundle a group of consecutive records that are considered for analysis. Formally, a time series window of length  $m$  is an ordered set of  $m$  consecutive samples (records) from the time series. Unique windows can be identified based on their first samples (or equivalently based on the timestamp of their first samples (see Figure 2.1).



**Figure 2.1:** Illustration for tumbling and sliding windows. Each black box represents a sensor value, and a group of  $m = 4$  consecutive are highlighted to represent a window.

Depending on the target analysis and what needs to be inspected within the time series data, two main methods to create time series windows are relevant: *Tumbling Window*, and *Sliding (aka Rolling, Hopping and Overlapping) Window*. Tumbling windows are created out of  $m$  samples over consecutive *disjoint* time intervals of the time series. For a time series with  $N$  samples,  $N/m$  disjoint tumbling windows can be created. A sliding window is derived by rolling a window of size  $m$  over the samples of time series. Therefore, a time series with  $N$  samples includes  $N - m + 1$  sliding windows where each two consecutive windows overlap by  $m - 1$  samples. Figure 2.1 illustrates an example of tumbling and sliding windows of size four over the samples of a time series of length 12.

### 2.1.6 Time Series Data Types and Formats

An important aspect of the systems dealing with storing, processing, and analyzing time series data is the data types and formats these systems use to represent and store time series data. This is related to the representation of the individual components in each record (tuple) of the time series. Specifically, this aspect relates to how the timestamp values and real-valued sensor data are represented.

For the representation of timestamps, often fixed point formats, specifically, the time stamps are typically stored as `INT64` data type (i.e., as `UNIX` timestamp). Other formats like `INT8`, `INT16`, and `INT32` are used less in the realm of time series analysis. As discussed before, in the case of uniform timestamps, a dense representation of timestamps can be achieved by just providing sampling rates.

Sensor values are often intrinsically real-valued, and therefore, floating point formats are often used to represent them. Specifically, the standard `IEEE 754` floating point format [`IEE19`], which was established as the technical standard for floating point arithmetic in the 80s and was revised recently in 2019, is used as the de-facto standard to represent sensor values in time series. This standard includes various data formats for representing, storing, and processing real-valued data.

`FP64`, double-precision, and `FP32`, single-precision, are the most widely used standard floating point formats and are mainly used in the context of scientific computing. `FP80` extended-precision format was used in Intel 8087 math co-processor [`IC80`] and is natively supported by x86 `CPUs` (its x87 instruction subset). However, this data format is not used today in data analytics and time series mining contexts. Many systems process and store monitoring data in `FP64` and `FP32` due to ease of use and wide support across various hardware platforms. Therefore, many time series data mining research works and frameworks typically exploit these formats and natively support them.

The use of `FP16`, half-precision, and `BFLOAT16`, Brain floating point Format, is not very common in scientific computing due to the small dynamic range; however, as this issue is less of a problem in machine learning applications, there is a growing trend in using these type of data for training neural networks. Also, Recently, `FP8` was introduced. However, its application is mainly limited to deep learning training and inference for the same reasons. `TF32` TensorFloat-32 [`Kha20`] is a reduced version of `FP32` with only 19 bits, where there are only ten precision bits instead of 23. `TF32` was first introduced together with `NVIDIA Ampere GPUs` [`Kha20`], and `NVIDIA tensor`



cores support this format natively. This format is also well suited for the precision requirements of Machine Learning workloads, not commonly used in the time series mining context.

### 2.1.7 Time Series Storage Layouts

Storing a single-dimensional time series is a straightforward issue as it requires storing (potentially encoding and compressing) a single data stream. Still, this issue becomes more complex when storing the main data stream together with timestamps. For example, the two streams, i.e., timestamps and the data series, can be stored independently, or the two streams can be laid out as a single stream of tuples of `<timestamp, sensor value>` where each record corresponds to a timestamp and the corresponding value of a target sensor (often real-valued). While the benefits of this scheme for storage might seem unintuitive, it can provide better performance for some time series analysis algorithms where both timestamp and sensor value might be loaded for processing [KRR<sup>+</sup>21].

Storage Layouts are typically more challenging for multi-dimensional time series. In this case, the collections of all sensor streams (dimensions) can be thought of as a table where each column corresponds to a sensor, and rows specify values of all sensors corresponding to a particular time stamp. While in some applications, storing the data row by row is efficient (e.g., in real time scenario on a data acquisition system), in other use cases, e.g., in data processing pipelines, column-by-column storage is more efficient. Such layouts also have direct implications on encoding, compression, and storage [KRR<sup>+</sup>21].

We highlight the importance and relevance of uniformity of time stamps in the context of time series storage layouts. Uniformity eliminates the need to store time stamps together with the data streams regardless of the dimensionality of the time series. It reduces the single-dimensional time series and, therefore, its storage to only deal with a single data stream, resulting in better storage efficiency. In the case of multi-dimensional time series, similar benefits are present for the storage. However, the challenges associated with the layout (e.g., column-wise vs. row-wise layouts) still remain.

In this work, we mainly focus on processing time series data in offline batch settings using algorithms with rather large computational complexity compared to I/O requirements. Therefore, we commonly use data stored in column-by-column format. For this, we often rely on custom binary formats suitable for parallel access with functionalities in common HPC environments.

### 2.1.8 Representative Examples

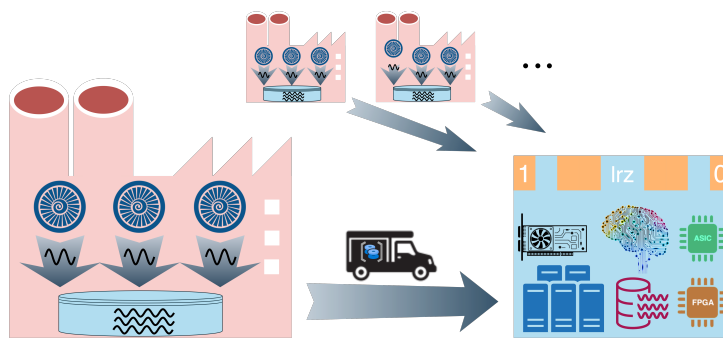
We use two representative domain examples to better demonstrate the time series concepts introduced earlier. At the same time, these examples serve as running examples throughout the thesis to describe the benefits of presented methods and optimizations. These two real-world examples motivated this work in the first place.

### 2.1.9 Use Case 1: Operational Data from Heavy-Duty Gas Turbines

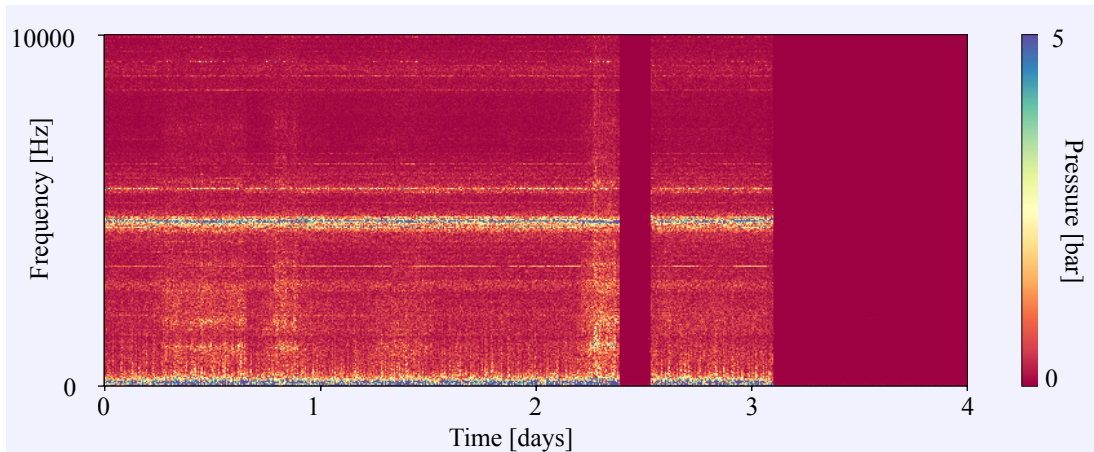
This first use case is motivated by the project Gas Turbine Optimization using Big Data (Turbo) [TS] funded by the Bavarian Research Foundation (2017-2020) to investigate and establish the capabilities for optimization of the operation of heavy-duty gas turbines through the explorative analysis of operational and dynamic sensor data. The central goal was to explore the effect of optimizing the operation of gas turbines to achieve more stable operation, better fuel consumption, and higher efficiency, and, consequently, to cause less air pollution for power generation. The main optimization approach in Turbo was based on gathering and analyzing data collected from the sensors and monitoring infrastructure deployed in power plants in Germany. In particular, heavy-duty gas turbines are equipped with a variety of sensors to monitor and enable feedback data to control their operation, which are leveraged for such explorative analysis. For example, the combustion process in a typical gas turbine is monitored with multiple sensors measuring dynamic pressure at a frequency of around 25 kHz. This data is often stored as single precision floating point numbers summing up to a huge volume of nine GB per day raw sensor data per sensor. Considering tens of power plant sites deploying heavy-duty gas turbines, the presence of multiple turbines per site, and the deployment of multiple sensors per turbine, the total amount of collected data easily exceeds *petabytes* per year. At that scale, even simple analysis and filtering on single or collections of multiple sensors is challenging and computationally demanding.

Conducting explorative data analysis on this amount of data requires the use of high-end powerful computer systems, similar to those available in HPC centers. Figure 2.2 illustrates the data collection from gas turbines in power plants for the purpose of explorative data analysis. After the collection of data from turbines, the data is transferred to a compute center for analysis.

The operational and dynamic sensor data collected in this use case are naturally in time series format. Figure 2.3 provides an example of a plot for pressure data from the combustion process inside the gas turbine. The data is presented as a spectrum in the frequency domain, and the pressure values are color-coded as a heat map representing the



**Figure 2.2:** Data generation and processing concepts in project Turbo. Time series data generated from monitoring of gas turbines is stored on disk and sent to a data center for processing.



**Figure 2.3:** Frequency spectrum of a sensor monitoring combustion process.

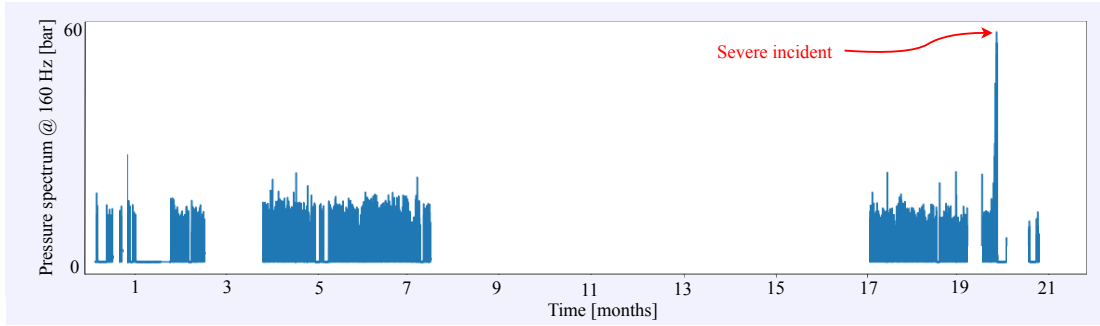
pressure amplitude at various frequencies over four days of operation. This visualization corresponds to 12 GB of data. While this visualization provides an overview and detailed picture of the pressure sensor values, analyzing this data is very challenging. In many cases, values at limited frequency bands are considered for analysis, which significantly reduces the processing, storage, and computational costs. In this case, the analysis/mining scenario can be mapped to single- or multi-dimensional time series analysis/mining problems.

In Figure 2.4, we illustrate an example of such mapping. We visualize the amplitude of the pressure signal at only the frequency of 160 Hz over the period of 21 months as a time series. This plot data also includes the data corresponding to a severe incident, which caused significant damage to the turbine, costing 10s of 1000s of Euros. By inspecting the data using time series mining and analysis techniques, we can gain insights, model, detect, and, with those, even prevent further incidents similar to the one depicted in Figure 2.4.

The monitoring data is also augmented with low-frequency data sampled at 1Hz, which includes the various conditional and operation data, for example, inlet temperature and pressure to the turbines. When considering 10s of such operational sensors per turbine and again 10s of sites, even this data quickly adds up to TBs of data per year.

While processing this data with advanced analytical methods can potentially bring valuable insights to the designers of gas turbines as well as to the operators of power plants, the processing of such amounts of data for fleets of turbines remains challenging. State-of-the-art approaches (before the start of the project [Turbo](#)) typically rely on various methods [[RHP16](#)] to estimate and project the analysis to a smaller problem. For example, in many cases, analysis is conducted on a limited time span, it relies on down-sampling, or it is based on filtering data on a specific frequency band [[RHP16](#)]. This is typically due to the limited computing resources available for such analyses since, traditionally, such analyses are conducted on workstations with limited memory and

## 2 Background and Technical Foundations

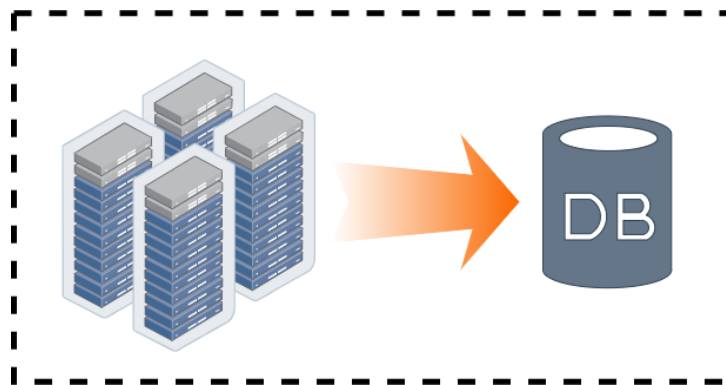


**Figure 2.4:** Timeline for the amplitude of combustion sensor at 160 Hz. The gaps in the data (e.g., after months seven until 17) are the result of missing monitoring data, which can have various reasons, including shutdown or maintenance.

computing. We introduce **HPC** systems into this picture and take advantage of the computing power of these systems. This can help to apply more complex time series analysis and mining methods at scale through **HPC**. With **HPC** in the picture, we increase the capabilities of analyzing larger amounts of data, enabling us to extract thorough insights from the data. Additionally, **gshpc** systems enable looking into time series data sampled at higher resolutions (higher frequencies) and potentially providing more details about the underlying physical processes (e.g., details of combustion processes). By employing various optimization techniques, as well as exploring reduced- and **mixed-precision** methods, the limits of computation capabilities are pushed forward.

### 2.1.10 Use Case 2: Operational Data Analytics in **HPC** Centers

The second use case is motivated by the **HPC** Operational Data Analytics (**ODA**) concepts [NSO<sup>+</sup>21] and cases in **DEEP** (“Dynamical **Exascale** Entry Platform - Software for **Exascale** Architectures”) project (2017-2021) funded by the European Com-



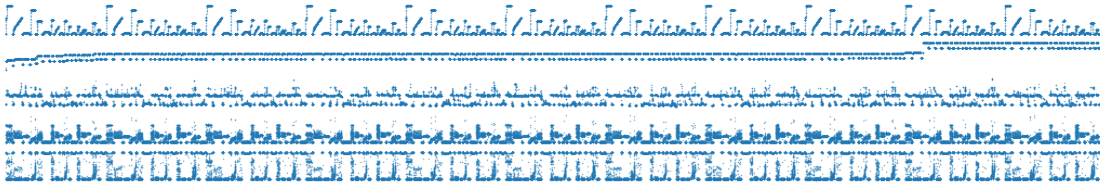
**Figure 2.5:** **HPC** monitoring (time series) data stored in a database for further (e.g., offline) processing.

mission [Gmb21a]. By leveraging the operational data collected through monitoring systems to optimize the operation of HPC systems (see Figure 2.5), operational data analytics has gained huge momentum in HPC sites over the last decade. The operation data includes various components of the systems, such as compute nodes, storage subsystems, and networking infrastructure, and it is not only limited to application performance data; its extent goes from the whole system to even full infrastructure operational data, including outbound power distribution units (PDUs) and the operation of cooling distribution units (CDUs). Such operational data are typically managed and stored by monitoring system tools, such as DCDB [NMA<sup>+</sup>19] and LDMS [AAB<sup>+</sup>14] in the shape of time series.

HPC operational data can be processed and analyzed to gain valuable insights into the details of the operation of the system, e.g., to identify potential performance bottlenecks or issues of the systems or applications running on the system. This consists of a variety of analytic tasks, such as workload characterization, fault detection, and diagnosis, with the intention of various optimizations of system operation, including performance optimization, energy efficiency improvement, better resource allocation, and job scheduling. One particular example of operational data analytics in HPC within the context of time series analysis is analyzing recurring patterns, trends, and anomalies in system behavior, as well as building predictive models to provide actionable insights to system administrators and HPC infrastructure operators.

Figure 2.6 presents an example of operational data from the HPCODAdataset [Net20], which is a collection of datasets acquired on production HPC systems and is representative of several real-world use cases in the field of Operational Data Analytics (ODA) for the improvement of reliability and energy efficiency. This data is presented and illustrated as a multi-dimensional time series.

In this particular example, we use the data collected from benchmark jobs on a single SuperMUC-NG compute node when running various CORAL-2 [VdSB<sup>+</sup>18] benchmarks, which is a benchmark suite developed in Collaboration of Oak Ridge, Argonne, and Livermore (CORAL) for joint-procurement activities among three of the US Department of Energy’s national laboratories. The data encompass a variety of performance counters, such as branch instructions, cache misses, and others, all collected using DCDB. Various shapes and patterns can be observed in the presented data in Figure 2.6. Analysis of such patterns can be crucial to gaining insights that can help in optimizing the operation



**Figure 2.6:** A sample from the operational data in HPCODA dataset [NMA<sup>+</sup>19] corresponding with the collection of various performance metrics from compute nodes of LRZ systems when running HPL.

of supercomputing centers. For instance, dynamic behaviors and patterns observed in user code execution can be analyzed, and with that, better statistics about applications running on HPC systems can be collected. Such statistics help to gather highly beneficial insights about the operation of HPC systems, e.g., to support both better operation of the systems and co-design of future systems. Another example is the analysis of energy consumption patterns by the HPC systems to achieve better scheduling schemes to achieve more efficient and cleaner operation.

### 2.2 Time Series Analysis and Mining

Analysis of time series is a broad field with a long history. The extent of methods in this field varies from statistical exploratory to recent machine learning-based prediction and modeling methods. Specifically for the context of this work, we can rely on the high-level classification of time series analysis methods presented by Pfeilschifter [Pfe19, Cha04] based on existing textbooks and literature reviews [Cha04, Kle15]: we can classify the existing methods into two non-mutually-exclusive classes of *traditional* methods which heavily rely on statistical methods and *data mining* methods with knowledge and insight discovery in perspective.

Traditional methods are often simple, intuitive, and popular. In many cases, these methods offer valuable insights into the data with minimal effort. These methods often consist of various basic approaches to the analysis of time series, including seasonalities, trends, and similarity characteristics. Analyzing these characteristics within time series provides a high-level *description* [JAV23] of the behavior embedded within data. Additionally, traditional methods include various classical explanatory and predictive [JAV23] modeling approaches, including curve fitting, ARIMA (autoregressive integrated moving average) modeling approach [BJ70]. The core idea in these types of approaches is to use sufficiently accurate models to explain the behaviors embedded in the data as well as to use these models to make predictions about the behavior of underlying systems generating the data.

More recent analysis methods, often considered as time series mining methods, include various schemes with knowledge discovery perspectives, often targeting large time series data volumes. These mining methods employ analysis methods and techniques from various disciplines, including statistics, but also extend to pattern recognition, machine learning, and neural networks [Col13]. The scope of time series mining is considered broader pattern discovery and can be extended to covering the discovery of hidden information or insights from time series data. Data mining methods and approaches often include analyzing correlations, similarities, irregularities, and patterns within time series data with the intention of gaining insights from the data that does not exist a priori [Col13]. Gaining insights from time series datasets can be the result of analyzing the raw time series sample points or other (potentially reduced) representations of time series datasets. Additionally, such methods are often used to conduct classical machine learning tasks such as clustering, classification, segmentation, and anomaly detection on time series data to achieve data-driven descriptive and predictive actionable insights



from data. Also, recently, more modern machine learning models and methods such as deep neural networks have been employed in time series mining [SZSF<sup>+</sup>19] to gain insights.

### 2.2.1 Time Series Similarity Indexing and Similarity Join

One of the primary goals of time series mining is to uncover previously unknown patterns in the time series datasets [Col13]. In many cases, time series mining approaches rely on analyzing *similarities* within time series data to achieve that. While naive statistical analyses of time series samples, as described in Section 2.2, can be utilized to provide valuable summaries and even insights into data, detailed similarity analysis of time series subsequences based on a particular similarity measure (e.g., correlation measure [FV17]) provides an even richer key representative similarity feature for analysis. Often, the similarity of time series subsequences based on correlation factor (or conversely distance metric, e.g., Euclidean distance) is used to characterize the patterns, similarities, or irregularities in the data. Various similarity measures [KK], such as Pearson’s Correlation (PC) [Kir08], and dissimilarity measures, including Euclidean Distance (ED) and Dynamic Time Warping (DTW) [Keo02].

The problem of characterizing and analyzing similarities is often represented by systematically searching and indexing time series subsequences (within one or multiple dimensions), as is common database management systems (DBMS). In the context of DBMS, this process is often referred to as similarity search, which is defined as the task of finding objects similar to a given query in a set of objects [Ech22]. Since in time series analysis and mining, often the target is to analyze the similarity of time series subsequences (i.e., local patterns), the set of these subsequences is considered as the relevant set of objects for indexing.

In DBMS, data structures known as *index* [Nat19] are constructed and maintained to serve the necessary information required to access and query data quickly and efficiently. The task of computing and maintaining such data structures is known as indexing. Specifically in the context of similarity mining, the index includes similarity association of the objects, e.g., pointers to the ID of similar objects.

Going beyond single queries, indexing similarities in time series, and maintaining a similarity index structure for time series subsequences can boost the query and analysis performance, as well as productivity for developing applications that rely on similarity association of time series subsequences. Similarity indexing in time series is referred to as creating and maintaining a similarity index for fast and efficient processing of similarity queries. Since such an index data structure typically encodes similarity correspondence among collections of time series subsequences, it enables various sophisticated analyses and mining tasks on the time series data and paves the path to gaining even more valuable insights, e.g., finding complex hidden recurring patterns, trends, and anomalies. Examples of such time series data mining tasks include motif discovery, novelty discovery, discord discovery, shapelet discovery, and so on. Typically, these tasks try to identify time series motifs (frequent patterns), outliers (rare patterns), or novelties (surprising patterns) as representative similarity features (primitives) that can be used for more

## 2 Background and Technical Foundations

complicated analyses. With such a time series similarity index in hands, acquiring these analysis primitive is straightforward, and therefore, computing similarity indexing renders various knowledge discoveries, data science, and machine learning tasks, e.g., classification, clustering, and semantic segmentation trivial [Keo23].

A commonly used index is for the nearest neighbor matching index, which stores the nearest neighbors association of objects (time series subsequences). Such indexing can boost analysis and data science that requires nearest neighbors in time series. This indexing provides a powerful view over the similarities among subsequences within a time series across multiple (typically two) time series [YZU<sup>+</sup>16]. For instance, we can consider an application that requires finding and processing a similarity query, i.e., to find the most similar (i.e., nearest neighbor) subsequence within the time series to the query. In such cases, in the presence of a pre-calculated nearest neighbor index, the task of calculating the similarity for the query falls down to a lookup.

Overall, time series similarity indexing and, specifically, nearest neighbor indexing play an important role as a key representative similarity index feature for time series analysis and mining.

So far, we established similarity indexing as a powerful approach to organize and represent similarities of time series subsequences within and across time series. Now, we introduce the *similarity joins* as abstract operations between time series to create and compute a so-called time series all-pair similarity index. Time series similarity join is defined as the operation between two input time series, i.e., a query and reference time series, in which the similarity of all possible pairs of subsequences within the two input time series is computed. To give a better picture, one can consider the task of computing a similarity matrix between all the subsequences of the two input time series and then finding the best matches (i.e., most similar) to each subsequence by scanning each row (or column). The output of this join operation is an index that identifies the similarity association of all pairs of the two input series. With this index at hand, finding subsequence similarity queries coming from the query time series, i.e., searching for similarities in the reference time series, boils down to simple lookups.

Time series join operations can be performed on two distinct input series, known as a cross-join, or between a time series and itself, which is known as *self-join*. In the case of the *self-join*, often, special treatments are considered to mitigate trivial similarity matches, i.e., retrieving a subsequence as the most similar to itself. In both cases, the amount of computation required to compute the index grows linearly with the size of the two input series (in the case of the cross-join). Therefore, in the case of *self-join*, the amount of computation grows quadratically with the size of the input series.

### 2.3 Foundations of Matrix Profile

One prominent, well-established method for time series join operation, and the corresponding indexing scheme is the *matrix profile* approach. A matrix profile provides a similarity index (profile) based on the nearest neighbors join operation between two input time series, often called query and reference time series. Having such a similarity index



(profile), one can lookup the nearest neighbor corresponding to a given subsequence in the query time series inside the reference time series. To highlight the importance and power of matrix profile, we can consider a use case where the reference time series is a well-known historical time series dataset that is used as a reference to help identify the motifs and patterns in an unknown query time series by leveraging the similarity indexing provided by matrix profile with cheap lookups.

The concept of similarity index (profile), in principle, also applies to multi-dimensional time series. However, for a multi-dimensional time series, the notion of similarity is generalized so that the analysis takes the correlation among subsequences in multiple dimensions into account. Also, multiple index (profile) structures are often required to represent the similarities for various dimensions, e.g., one index for each dimension.

Since matrix profile is the core time series mining concept in this work, we will provide a deeper background on it. We start by introducing the term *matrix profile* as the similarity indexing join operation for time series. Therefore, depending on the context, we often use the term *matrix profile* to refer to the similarity join operation. Also, we use the term *matrix profile computation* to refer to the calculations and operations associated with conducting the join operation that are required to be executed to get the similarity index. On the other hand, the outputs of the matrix profile computation are two index data structures called *matrix profile* and *matrix profile index*. Therefore, in some contexts, we use the term *matrix profile* to refer to these output index structures as well.

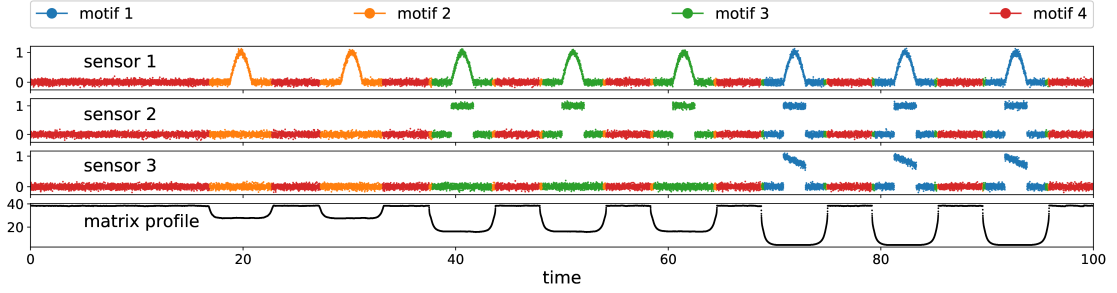
As discussed before, the join operation, therefore computing the matrix profile, between the input series, is associated with a distance matrix, which includes the distance among all the subsequence pairs of the input series. In fact, that is where the term *matrix* in the name “matrix profile” is coming from. The whole idea behind the matrix profile approach is to summarize a complete distance matrix (or equivalently, correlation matrix) of all the subsequences of input time series into two index data structures:

- *the matrix profile* (denoted as  $P$ ), which is a real-valued vector encoding the distance of a subsequence to its nearest neighbor, and
- *the matrix profile index* (denoted as  $I$ ), which is an indexing vector storing pointers to the nearest neighbor of a subsequence to identify the location (index) of its nearest neighbor.

This definition can be applied to all cases of cross-join and [self-join](#) in single and multi-dimensional time series mining. Based on this definition, the matrix profile is associated with the nearest neighbors join operation between input series. Therefore in this work, we refer to the task performing this nearest neighbor join operation, i.e., calculating the matrix profile and matrix profile index ( $P$  and  $I$ ), as “matrix profile computation” or “computing the matrix profile”.

We provide an illustration for the matrix profile by presenting a simple analysis scenario in Figure 2.7. In this illustration, we intend to visually demonstrate the matrix profile and show how it can be used for semantic segmentation of a multi-dimensional time series. The goal of such segmentation is to partition the time series into regions

## 2 Background and Technical Foundations



**Figure 2.7:** Illustration of matrix profile and its use for semantic segmentation of a multi-dimensional time series by finding motifs through matrix profile analysis.

(segments) where each region consists of similar structures. Our intention here is to show the capabilities of the approach without binding it to a specific domain, and therefore, we present a synthetic example. For more illustrative real-world examples, we refer to Yeh et al. [YHK16, YZU<sup>+</sup>16] and Gharghabi et al. [GYD<sup>+</sup>].

In our synthetic example, we consider a three-dimensional time series representing three sensors, each providing 20 000 samples. The first carries  $\sin(x)$ -pulse starting at +20 seconds. The second carries square-pulse starting at time +40 seconds. Finally, the third sensor carries a sawtooth-pulse starting at +70 seconds; all wavelets are recurring every 10 seconds. This introduces unique phases (segments) in the presented sensor data during which the structure of sensor data is *unique*. Here, by structure, we refer to the patterns embedded in the sensor data in each dimension separately (e.g., the sine waves on the first sensor) or the combination of the patterns embedded in all three dimensions (e.g., the combination of the sine wavelet shapes on the first sensor together with the flat shapes on the other two sensors, i.e., highlighted in orange). We call the multi-dimensional structures (here three-dimensional structures) motifs. Motifs are patterns with a unique correlation structure among all the sensors that might recur over time. For instance, we observe four motifs in Figure 2.7, each highlighted with a separate color (red, orange, green, and blue). Note that when looking at individual sensor data, we might find different (single-dimensional) motifs. For instance, by only looking at the first sensor, we only detect two motifs (sine wavelets and flat regions). However, the three-dimensional motifs consider the structures of all three sensors together. The multi-dimensional motifs detected are also closely related to semantic segmentation of the time series, where the presented time series can be segmented into four regions with similar sensor structures.

We can conduct a [self-join](#) matrix profile to identify the similarity structure of three-dimensional subsequences<sup>1</sup>. The matrix profile is computed by constructing the distance matrix for all the three-dimensional subsequences in the time series and finding the best match (i.e., the nearest neighbor) for each individual three-dimensional subsequence. We illustrate the matrix profile at the bottom of Figure 2.7 in black color. At each

<sup>1</sup>The subsequence length, which is a parameter in matrix profile analysis and will be introduced later in this chapter, is set to four seconds.

point, the value of the matrix profile corresponds to the distance of the three-dimensional subsequence to its best match in the series. We observe that the matrix profile highlights and distinguishes the four phases by summarizing the correlation structure among the sensors in the time series.

To further illustrate the power of the matrix profile in practice, we feed the computed matrix profile to a clustering method to find distinct groups (clusters) with similar matrix profile values. This helps to identify phases of the time series according to the segments on the matrix profile with similar values. Specifically, we use the k-means algorithm with  $k$ , the number of groups set to four, to find four groups of values of this matrix profile. This helps to group the corresponding classes as similar motifs based on their distances in the matrix profile (see Figure 2.7, color-coded time series segments). This analysis results in meaningful segments corresponding to the respective phases (and motifs) in the original data.

Contributions of this work, which are introduced in the next chapters, heavily rely on understanding the details of matrix profile computation. Therefore, we continue this chapter by introducing detailed mathematical foundations of the matrix profile.

### 2.3.1 Matrix Profile Notations and Formulations

We mainly follow the notations and formulations introduced in the work of Yeh et al. [YZU<sup>+</sup>16], Raoofy et al. [RKY<sup>+</sup>20], and Pfeilschifter [Pfe19]. We first start by introducing the terminology for the single-dimensional time series denoted as  $T$ .

**Definition 1.** A time series,  $T$ , is a sequence of real-valued numbers  $t_i \in \mathbb{R} : T = t_1, t_2, \dots, t_n$ , where  $n$  is the number of records or the length of  $T$ .

As discussed earlier in Section 2.2.1, in the context of time series similarity join matrix profile, we rely on comparing time series subsequences. These are local chunks of time series, also known as time series segments. By definition, subsequences are also time series themselves. These subsequences are derived from rolling a window of a certain size (subsequence length) over the time series. Especially in the context of matrix profile, only overlapping subsequences are considered.

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

Moreover, the matrix profile is commonly computed based on the **z-normalized** Euclidean distances (standardized distances) among subsequences of a time series. This way, the subsequences are normalized in a scale-invariant fashion (i.e., subsequences are normalized by their standard deviation and shifted to have zero means).

**Definition 3.** The **z-normalized** Euclidean distance between two subsequences  $T_{i,m}^q$  and  $T_{j,m}^r$  of length  $m$ , denoted as  $\delta(T_{i,m}^q, T_{j,m}^r)$ , is defined as the Euclidean distance  $\delta : m \times m \rightarrow \mathbb{N}_0^+$  between the two subsequences after each is **z-normalized**, i.e., rescaled to zero-mean and unit-variance.

## 2 Background and Technical Foundations

While it is possible to extend the definition of distance matrix to metrics beyond the **z-normalized** Euclidian distance, we still rely on this metric in the spirit of the original work introducing the matrix profile concept [YZU<sup>+</sup>16], as well as to simplify correctness and verify the accuracy of results. Additionally, the usage and advantage of **z-normalized** distances in the context of time series similarity indexing (and therefore matrix profile) is empirically justified in existing studies [KK, WMD<sup>+</sup>13, RCM<sup>+</sup>12].

**Definition 4.** Let  $T^r$  and  $T^q$  be two time series of lengths  $n \in \mathbb{N}$  and  $l \in \mathbb{N}$ , which can potentially be identical. Given a query length  $m \in \mathbb{N}, 2 < m \leq \min(n, l)$  the distance matrix of size  $(l - m + 1 \times n - m + 1)$  is defined as  $D^{r,q} = [d_{i,j}]$  where each element of this matrix is determined as  $d_{i,j} = \delta(T_{i,m}^q, T_{j,m}^r)$ .

This is the formal definition for the distance matrix that we discussed in Section 2.2.1.

In more detail, the element  $d_{i,j}$  sitting on  $i$ -th row and  $j$ -th column of the distance matrix  $D^{r,q}$  corresponds to the **z-normalized** Euclidean distance ( $\delta$ ) between  $i$ -th subsequence in  $T^q$  to the  $j$ -th subsequence in  $T^r$ . This implies that the  $i$ -th row of the distance matrix contains the distances of  $i$ -th subsequence in  $T^q$  to all the subsequences in  $T^r$ . Therefore, by scanning the  $i$ -th row of the distance matrix, one can identify the subsequence in  $T^r$  with minimal distance to  $i$ -th subsequence in  $T^q$ . This particular subsequence in  $T^r$  represents the best match (most similar or nearest neighbor) to  $i$ -th subsequence in  $T^q$ .

The distance matrix  $D^{r,q}$ , contains all pairwise distances between subsequences from  $T^r$  and  $T^q$ . Consequently, the distance matrix can be used to identify the nearest neighbors for all subsequences  $T^q$  in  $T^r$ .

We now define time series similarity join (or nearest neighbor join) based on the distance matrix.

**Definition 5.** Given a query length  $m$ , the time series similarity join operation between  $T^r$  and  $T^q$  is defined as the joining (processing) the input series to get the set of  $l - m + 1$  subsequence index tuples  $(i, j)$ , where  $j$  refers is the index of a subsequence in  $T^r$  that is nearest neighbor of the subsequence  $i$  in  $T^q$ . This join operation is denoted as  $T^r \bowtie_{1nn}^m T^q$ .

The distance matrix  $D^{r,q}$  can be used in computation of  $T^r \bowtie_{1nn}^m T^q$ . In particular,  $T^r \bowtie_{1nn}^m T^q[i] = \min_j D^{r,q}[i, j]$ .

Note that in the general case,  $D^{r,q}$  is not equal to  $D^{q,r}$ . and therefore  $T^r \bowtie_{1nn}^m T^q$  is not the same as  $T^q \bowtie_{1nn}^m T^r$ . However, in the special case, where the two input time series are the same, i.e.,  $T^r = T^q = T$ , the distance matrix  $D$  is symmetric  $D = D^{q,r} = D^{r,q}$ . This property can be used to reduce computation costs in such special cases. This special case is called the *self-join* operation ( $T \bowtie_{1nn}^m T$ ) and is a common setup in many of the matrix profile computation scenarios and problems in literature. Note that, in this case, the diagonal entries  $d_{i,i}$  are equal to 0, as they measure the **z-normalized** distance between a subsequence and itself. These cases are considered trivial matches and should be ignored from the computation. Furthermore, close to the diagonal, typically, all distance values are small and are also considered trivial matches and are ignored in

similarity computation since the similarity of windows that are in close affinity typically are not interesting. We denote these trivial matches with  $E_i$ . For subsequence  $i$  in  $T^q$ ,  $E_i$  is the set of indices ( $j | j \in [1; n - m + 1]$ ) of all the subsequences in  $T^r$  that are assumed or known to be trivial matches a priori.

With all the above definitions provided, we can now move forward and introduce the formal definitions of the **matrix profile** and **matrix profile index**.

**Definition 6.** Let  $T^r$  and  $T^q$  be the reference and the query time series of lengths  $n \in \mathbb{N}$  and  $l \in \mathbb{N}$ , which can potentially be identical (i.e.,  $T^r = T^q$ ). Let  $E_i \subset [1; n - m + 1]$  denote the set containing the indices of all trivial matches (or any matches that need to be excluded for any reason) of the subsequence  $T_{i,m}^q$  in  $T^r$ . The **matrix profile**  $P^{r,q}$  is defined as the vector of real valued distances  $P^{r,q} = (p_1, p_2, \dots, p_{l-m+1})$  where  $p_i = \min_j (d_{i,j}) \quad | \quad j \in [1; n - m + 1] \setminus E_i$ .

In simple words, the matrix profile is a vector whose elements represent the minima of the rows in the distance matrix. This translates to the following: the entry  $p_i$  of the matrix profile is the distance of subsequence  $T_{q_i,m}$  to its nearest neighbor among all the subsequences of  $T^r$  with length  $m$ .

The definition of matrix profile index follows the definition of matrix profile with a small modification. For the matrix profile index, we compute *argmin* in each row of the distance matrix instead of *min*. *argmin* is defined as the operation that, when applied to a vector, returns the integer value, *index*, in which the vector has the minimum value (i.e., it returns the location of the minimum value in the vector).

**Definition 7.** Let  $T^r$  and  $T^q$  be the reference and the query time series of lengths  $n \in \mathbb{N}$  and  $l \in \mathbb{N}$ , which can potentially be identical (i.e.,  $T^r = T^q$ ). Let  $E_i \subset [1; n - m + 1]$  denote the set containing the indices of all trivial matches (or any matches that need to be excluded for any reason) of the subsequence  $T_{i,m}^q$  in  $T^r$ . The **matrix profile index**  $I^{r,q}$  is defined as the vector of integers  $I^{r,q} = (q_1, q_2, \dots, q_{l-m+1})$  where  $q_i = \underset{j}{\operatorname{argmin}} (d_{i,j}) \quad | \quad j \in [1; l - m + 1] \setminus E_i$ . In the special case of several minimizer indices  $j$ , the smallest one is chosen.

In simple words, the matrix profile is a vector whose elements represent the minima of the rows in the distance matrix. The matrix profile entry at location  $i$  ( $q_i$ ) includes the index,  $j$ , associated with the nearest neighbor subsequence  $T_{j,m}^r$  for the subsequence  $i$  in  $T^q$ .

With the definitions presented for the matrix profile and matrix profile index, we reiterate the fact that the matrix profile concept is built on top of the nearest neighbor similarity indexing and provides a similarity index for analyzing time series.

In the case where the two input series are equal, i.e.,  $T^r = T^q = T$ , most implementations use a homogeneous and symmetric exclusion zone ( $E$ ). Here, symmetry refers to the exclusion of the  $e$  samples from both left and right of the index  $i$ , i.e.,  $E_i = [i - e; i + e]$ . Also, homogeneity refers to using the same value of  $e$  for all the reference indices  $i$ . In particular, most implementations exclude trivial matches  $E_i = [i - m/4; i + m/4]$  or

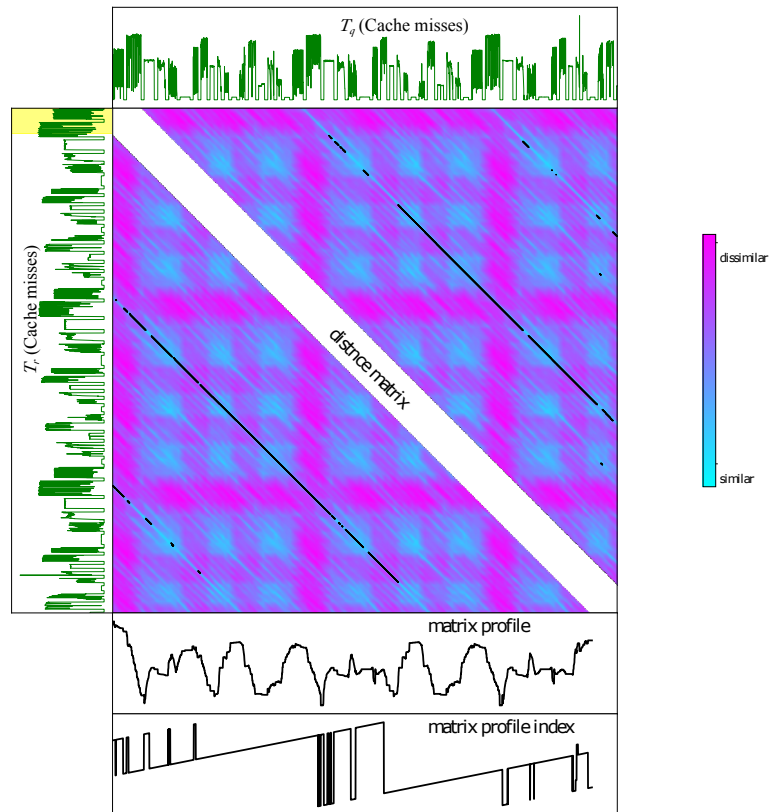
## 2 Background and Technical Foundations

$E_i = [i - m; i + m]$ , where an exclusion zone equal to the size of subsequence length  $m$  or a quarter of it is chosen. In such cases, often, the length of the exclusion zone is defined by a parameter  $e$ , which in the above cases is set to either  $m/4$  or  $m$ . This exclusion setting ignores the fact that, in theory, the size of the trivial matching zone is data-dependent, as illustrated by Yeh et al. [YHK16].

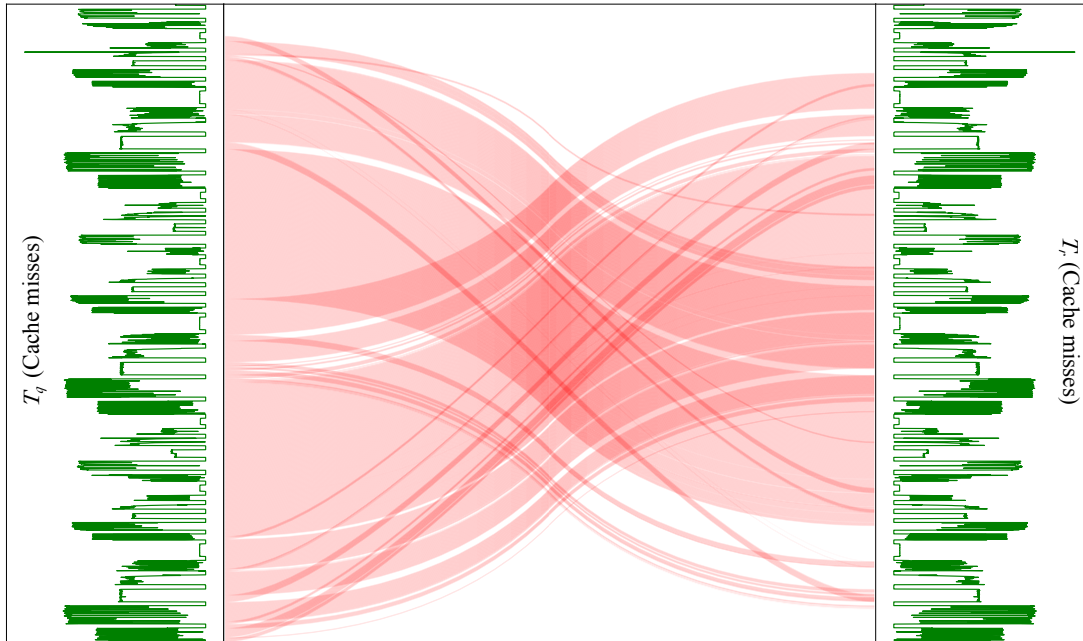
However, this is still a practical approach and is aligned with the fact that, in practice, the matches in close affinity are too trivial to be interesting. Besides, the studies empirically show that this choice is sufficient and safe [YZU<sup>+</sup>16, YHK16].

### 2.3.2 Matrix Profile Illustration – Operational Data from HPC centers

We revisit the second use case introduced in Section 2.1 and illustrate the above definitions with the intention of creating a better intuition about matrix profile and its computation. In this illustration, from the HPCODA dataset, we use the application classification segment corresponding to labeled operational data corresponding to CORAL-2 benchmark suite [VdSB<sup>+</sup>18] runs (We introduced this benchmark suite in Chapter 2). We use the cache miss values on a single node and plot it as a time series (plotted in



**Figure 2.8:** Illustration of the distance matrix, matrix profile, and matrix profile index.



**Figure 2.9:** Illustration of similarity associations between two time series when the matrix profile index is used to create the red arcs.

green on the top and left side in Figure 2.8). We use the same time series for both the reference and query time series (*self-join*). The distance matrix whose entries correspond to *z-normalized* Euclidean distance of subsequence pairs in the two input series is also illustrated. The matrix is color-coded with values of the distance matrix such that similar subsequence pairs have blue colors and dissimilar ones have red colors.

The window size considered in this case is depicted with the color yellow in the graph. For the purpose of this text and generating intuitive visualizations, we use fairly large window sizes, while in general, smaller window sizes are equally valuable and insightful.

While the matrix profile concept is general in the sense of applying to two distinct time series, here, for the sake of illustration, we consider the case of *self-join*. As required in the *self-join* case, we apply an exclusion zone to exclude trivial matches along the diagonals (entries in the distance matrix with white color).

As discussed before, the matrix profile can be computed by finding the minimum values along the columns of the distance matrix. We highlight these minimum values with black on the distance matrix. The matrix profile is computed using these minimum values and their locations. We illustrated the matrix profile (the minimum value and its location along the columns of the matrix) at the bottom in Figure 2.8.

While interpreting the plotted matrix profile and matrix profile index seems unintuitive at first, they provide valuable insights: for instance, minimum values on the matrix profile graph correspond to subsequences in  $T^q$  that have a similar counterpart in  $T^r$ . For better intuition for the matrix profile index, in Figure 2.9, we provide a visualization



## 2 Background and Technical Foundations

for the corresponding similarity indices computed in the previous example. We illustrate the two input time series on the two sides. We use the matrix profile index to create association lines between the query and reference series. In simple words, we follow the values of the matrix profile index to find the location of the nearest neighbors of subsequences of  $T^q$  in  $T^r$ . We use a so-called arc graph to represent these associations, and the illustrated arcs show the nearest neighbors association between  $T^q$  and  $T^r$ , which helps to visually grasp the similarities of the two time series. In particular, this illustration helps to visually associate the cache miss patterns (subsequences) in the query input time series  $T^q$  to the corresponding matches in the reference input time series  $T^r$  using the red arcs.

### 2.3.3 Multi-Dimensional Matrix Profile – Notations and Formulations

Next, we introduce notations and formulations for multi-dimensional time series and extend the concept of matrix profile for this type of time series. In particular, we use the terminology introduced by Yeh et al. [YZU+16] and Raoofy et al. [RKY+20].

**Definition 8.** A multi-dimensional time series  $T \in \mathbb{R}^{d \times n}$  is a set of coevolving time series,  $T = [T^{(1)}, T^{(2)}, \dots, T^{(d)}]$  where,  $d$ , denotes the dimensionality of  $T$  and,  $n$ , represents the length of  $T$ . Each  $T^{(x)} \in \mathbb{R}^n$  is a single-dimensional time series and is associated with one dimension.

**Definition 9.** A multi-dimensional subsequence  $T_{i,m} \in \mathbb{R}^{d \times m}$  of a multi-dimensional time series  $T$  is a multi-dimensional time series of length  $m$  starting from position  $i$  in  $T$ :  $T_{i,m} = [T_{i,m}^{(1)}, T_{i,m}^{(2)}, \dots, T_{i,m}^{(d)}]$ .

The above definitions are a straightforward generalization of the Definition 1 and Definition 2 provided in the single-dimensional case. One can think mining subsequence similarities with such definitions for multi-dimensional time series might be a straightforward extension of single-dimensional cases. However, the extension of the matrix profile concept case is more general to consider similarities across different dimensions as well as similarities of multi-dimensional subsequences. This means that the multi-dimensional matrix profile case considers the matching of subsequences with dimensionalities smaller than  $d$ , e.g., 1-dimensional, 2-dimensional,  $\dots$ , and  $d$ -dimensional subsequences where the dimensions of the subsequence span only a subset of the  $d$  dimensions. The reason for this design in the case of multi-dimensional matrix profile case is that typically, more than only  $d$ -dimensional subsequence matches are of interest in practice. In fact, as discussed in detail in the work of Yeh et al. [YZU+16], using  $d$ -dimensional matrix profile analysis for motif discovery is generally guaranteed to fail, and therefore, in general, only a subset of all dimensions must be used for multi-dimensional motif discovery.

**Definition 10.** A sub-dimensional subsequence  $T_{i,m}(X) \in \mathbb{R}^{k \times m}$  is a multi-dimensional subsequence for which only a subset of  $d$  dimensions in  $T$  is selected.  $k \leq d$  is the number of dimensions included in  $T_{i,m}(X)$ , and  $X$  is an indicator vector with the cardinality of  $k$  ( $\|X\|_0 = k$ ) which shows the dimensions that are included.



This definition allows generalizing the distance matrix and matrix profile concepts such that the distances among all sub-dimensional subsequences are considered in the analysis. In other words, this definition allows us to consider  $k$ -dimensional distances between  $k$ -dimensional sub-dimensional subsequences in the case of multi-dimensional time series. The definition of multi-dimensional matrix profile introduced by Yet et al. [YKK17] covers similarities for all the 1-dimensional, 2-dimensional,  $\dots$ , and  $d$ -dimensional sub-dimensional subsequences. This introduces a combinatorial search space to mine all the sub-dimensional spaces.

**Definition 11.** *The  $k$ -dimensional distance measure,  $\delta^{(k)}$ , represents the distance between two multi-dimensional subsequences  $T_{i,m} \in \mathbb{R}^{d \times m}$   $T_{j,m} \in \mathbb{R}^{d \times m}$  by using only the  $k$  out of  $d$  dimensions, where  $\delta^{(k)}$  is the minimum distance value for all possible combinations of  $k$  out of  $d$  dimensions (i.e., for any vector  $X$ ). Formally,  $\delta^{(k)}(T_{i,m}, T_{j,m}) = \min_X \delta(T_{i,m}(X), T_{j,m}(X))$ , where  $\|X\|_0 = k$ .  $\delta$  is usually considered to be the  $z$ -normalized Euclidean distance.*

This definition provides the distance between sub-dimensional subsequences. This helps to represent similar sub-dimensional subsequences as the “best matching” subsequences (with minimal distance) in the combinatorial space. This notion of distance allows us to mine similarities in all the  $k$ -dimensional sub-dimensional subsequences in combinatorial space of sub-dimensional subsequences.

**Definition 12.** *Let  $T^r$  and  $T^q$  be two  $d$ -dimensional time series of lengths  $n \in \mathbb{N}$  and  $l \in \mathbb{N}$ , which can potentially be identical. Given a query length  $m$ , multi-dimensional time series similarity join operation between  $T^r$  and  $T^q$  is defined as computing sub-dimensional subsequence pairs with  $k \leq d$ , when the distance is computed by using the  $k$ -dimensional distance function. Formally, it translates to computing the pairs  $(T_{i,m}(X^k), T_{j,m}(X^k)) | \forall i \leq n, j \leq l, \forall X^k$ , where  $T_{j,m}(X^k)$  is the nearest neighbor of  $T_{i,m}(X^k)$  in  $T^r$  assuming the indicator vector  $X^k$  considered all combinatorial sub-dimensional subsequences with dimensionality of  $k$ . This join operation is denoted as  $T^r \bowtie_{1nn}^k T^q$ .*

With this definition, multi-dimensional time series similarity join corresponds to finding the matches in the combinatorial space of sub-dimensional subsequences.

**Definition 13.** *Let  $T^r$  and  $T^q$  be two  $d$ -dimensional time series of lengths  $n \in \mathbb{N}$  and  $l \in \mathbb{N}$ , which can potentially be identical. The multi-dimensional **matrix profile**  $P^{r,q}$  is defined as the set of  $d$  (single-dimensional) matrix profiles  $P^{r,q} = [\rho_1, \rho_2, \dots, \rho_d]$ , each corresponding to the individual sub-dimensions, where  $\rho_k$  is the vector of  $k$ -dimensional distances of all pairs computed with  $T^r \bowtie_{1nn}^k T^q$ .*

**Definition 14.** *Let  $T^r$  and  $T^q$  be two  $d$ -dimensional time series of lengths  $n \in \mathbb{N}$  and  $l \in \mathbb{N}$ , which can potentially be identical. The multi-dimensional **matrix profile**  $P^{r,q}$  is defined as the set of  $d$  (single-dimensional) matrix profiles Indices  $I^{r,q} = [\iota_1, \iota_2, \dots, \iota_d]$ , each corresponding to the individual sub-dimensions, where  $\iota_k$  is the vector of indices of all  $k$ -dimensional pairs computed with  $T^r \bowtie_{1nn}^k T^q$ .*

## 2 Background and Technical Foundations

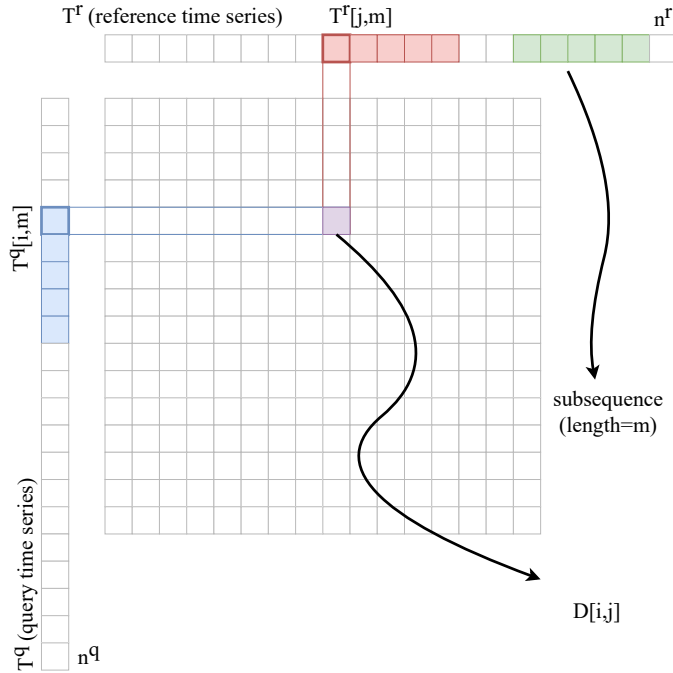
**Definition 15.** A  $k$ -dimensional matrix profile subspace  $S \in \mathbb{N}^{k \times (n-m+1)}$  is a multi-dimensional meta series that stores the  $k$  dimensions associated with subsequences computed in  $T^r \bowtie_{1nn}^m T^q$ .

With this definition, the matrix profile approach offers a powerful data structure to encode multi-dimensional motifs and similarity indices within the sub-dimensions of time series that may not be apparent when considering each dimension individually. This enables the discovery of complex hidden structures across different sub-dimensions, which offers valuable insights into the dynamics and interdependencies of sub-dimensions.

### 2.3.4 Background on Matrix Profile Computation and its Costs

We start by introducing the task of matrix profile computation in the single-dimensional case. The multi-dimensional case will be covered in the following chapters. We provide a Pseudocode 1 adapted from the original work of Yeh et al. [YZU<sup>+</sup>16] to show how matrix profile can be computed naively and demonstrate the computational costs involved. We work on the cross-join case with two input time series,  $T^r \in \mathbb{N}^{n^r \times 1}$  as the reference and  $T^q \in \mathbb{N}^{n^q \times 1}$  as the query. Also, we consider a subsequence length of interest of size  $m$  (see Figure 2.10).

Pseudocode 1 starts with a loop nest to compare all the subsequence pairs in  $T^r$  and  $T^q$ . We iterate ( $i$  loop) through the query subsequences ( $T^q[i, m]$ ), in Line 1. We then iterate through ( $j$  loop) the reference subsequences ( $T^r[j, m]$ ) in Line 2. In Line 3,



**Figure 2.10:** Illustration the distance matrix and naive matrix profile computation.

we compute the **z-normalized** Euclidean distance between subsequences  $T^q[i, m]$  and  $T^q[j, m]$ . Overall, Lines 1-5 compute the whole distance matrix. With the distance matrix at hand, for each row of the distance matrix (loop in Line 6 and Line 6), we compute the **min** and **argmin** and store them to output vectors at location  $i$ , i.e.,  $P^{r,q}[i]$  and  $I^{r,q}[i]$ . This computation matches the Definition 6 and Definition 7, and therefore, the  $i$ -th element of  $P^{r,q}$  corresponds with the distance of  $T^q[i, m]$  to its most similar subsequence in  $T^r$ . Likewise, the  $i$ -th element of  $I^{r,q}$  corresponds with the index of the most similar subsequence to  $T^q[i, m]$  in  $T^r$ .

Pseudocode 1 and Figure 2.10 illustrate that the main cost in the computation of the matrix profile is related to calculating the distances of subsequence pairs, i.e., the elements of the distance matrix. For time series  $T^r$  and  $T^q$  of lengths  $n_q \in \mathbb{N}$  and  $n_r \in \mathbb{N}$ , this corresponds to computing a distance matrix of size  $(n_q - m + 1) \times (n_r - m + 1)$  (i.e.,  $\mathcal{O}(n_q \times n_r)$ ). Therefore, in the case of **self-join** (i.e.,  $T^r = T^q$ , and  $n_q = n_r$ ), the computational cost is quadratic,  $\mathcal{O}(n^2)$ .

For the naive (brute-force) approach described above, the computation associated with each element of the distance matrix includes a **z-normalized** Euclidean distance calculation of cost  $\mathcal{O}(m)$ . Therefore, the overall computational cost in the case of brute force computation described above is  $\mathcal{O}(n^2 \times m)$ . Note that these costs also match the expected costs associated with the family of all-nearest-neighbor problems (nearest-neighbor joins). In fact, the matrix profile is a special case for the general all nearest-neighbors problems for time series, and therefore, their brute force computational costs match.

Overall, the core part of matrix profile computation is to calculate elements of the distance matrix  $\mathcal{D} \in \mathbb{R}^{(n_q-m+1) \times (n_r-m+1)}$ . Therefore, state-of-the-art methods for matrix profile computation often rely on optimizing the computation of the distance matrix. For instance, one promising front is to use mathematical formulations such as streaming dot product formulation [Nes23] with improved computational costs for distance compu-

---

**Pseudocode 1** The naive scheme to matrix profile computation for single dimensional time series.

---

**Input:** The reference and query time series  $T^r \in \mathbb{N}^{n_r \times 1}$  and  $T^q \in \mathbb{N}^{n_q \times 1}$ , and subsequence length  $m$ .

**Output:** The matrix profile  $P^{r,q}$  and index  $I^{r,q}$ , associated with  $T^r \bowtie_{1:n_r}^k{}^m T^q$ .

---

```

1: for  $i \leftarrow 0$  to  $n_q - m + 1$  do
2:   for  $j \leftarrow 0$  to  $n_r - m + 1$  do
3:      $D[i, j] = \delta(T^r[i, m], T^q[j, m])$      $\triangleright$  compute the distance between query ( $j$ ) and reference ( $i$ )
4:   end for
5: end for
6: for  $i \leftarrow 0$  to  $n_q$  do
7:    $P^{r,q}[i] \leftarrow \min(D[i, :])$            $\triangleright$  find the minimum distance for subsequence query  $i$ 
8:    $I^{r,q}[i] \leftarrow \operatorname{argmin}(D[i, :])$      $\triangleright$  find the argmin for subsequence query  $i$ 
9: end for

```

---

tation that are optimized for computing the (**z-normalized**) Euclidean distance for time series. The idea behind such formulations is to identify the common factors contributing to distance computation and avoid their redundant computation. For instance, such ideas can bring in math formulations to compute the distance of two subsequences when the distance between their immediate neighboring subsequences is evaluated. This can effectively eliminate the  $\mathcal{O}(m)$  computational cost for each element in the distance matrix. With such enhanced mathematical formulations, the computation of the matrix profile can be reduced by a factor of  $\mathcal{O}(m)$ . Still, the overall computational cost, even when leveraging such optimizations, remains at  $\mathcal{O}(n^2)$ . This is slightly better than the costs for the naive approach and still grows quadratically with the number of records ( $n$ ) in the input time series.

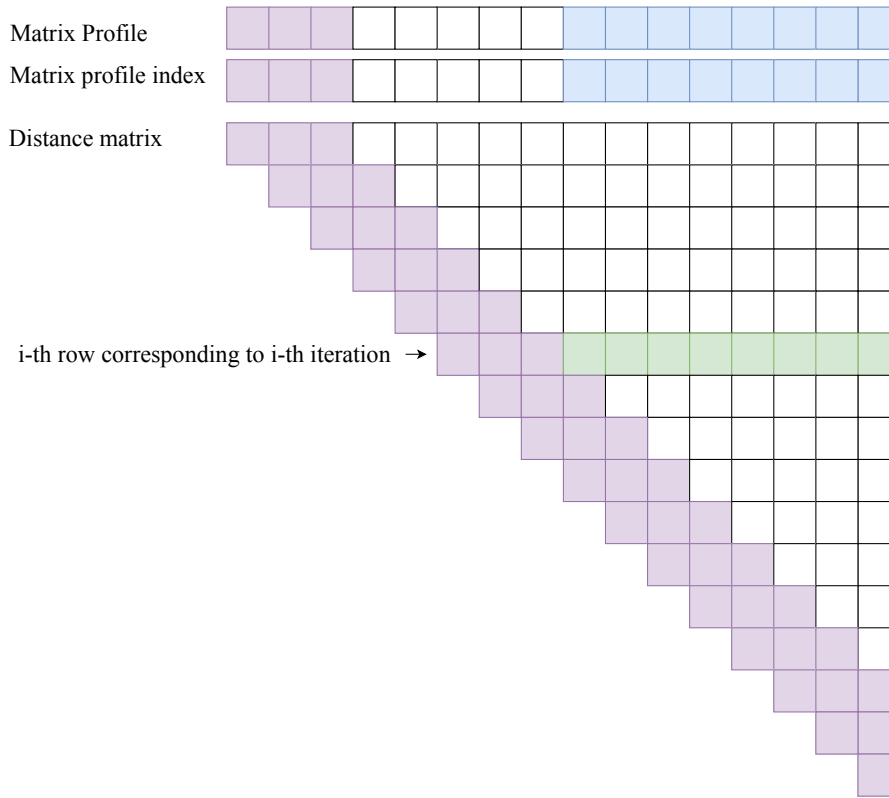
Another front in optimizing matrix profile computation is to avoid huge memory consumption demands. The memory utilization for the naive approach is  $\mathcal{O}(n^2)$  to store the distance matrix. This quadratic demand for storing the matrix in memory can become punishing in various hardware. Therefore, state-of-the-art methods avoid storing the full distance matrix. These approaches rely on computing the distance matrix row-by-row (or equivalently column-by-column) and updating the best so-far matrix profile and matrix profile index computation. This is achieved by fusing the two loops in Lines 1 and Line 6 in Pseudocode 1. This eliminates the need to store the full distance matrix and only requires storing one row (or column) of the distance matrix at a time. This approach can be interpreted as an iterative scheme in which the values of the matrix profile and matrix profile index are iteratively updated as the rows/columns of the distance matrix are progressively computed. We discuss such iterative approaches in more detail in the following section.

### 2.3.5 Iterative In-Place Matrix Profile Computation with Enhanced Distance Formulation

We describe the **STOMP** (stands for Scalable Time series Ordered-search Matrix Profile) method for computing the matrix profile, which follows the kind of iterative scheme to compute the matrix profile described in the previous subsection. We start with **STOMP** since it can be considered the mother of similar iterative methods called **STOMP<sub>opt</sub>** in the literature. We also rely on this approach for most parts of our work; however, for convenience, we refer to it as **STOMP**.

This approach leverages an iterative distance matrix computation. This means that the computation of the distance matrix happens iteratively, where in each iteration, the resulting matrix profile and matrix profile index are updated (see the description in the previous section again). Additionally, **STOMP** leverages in-place distance matrix computation. This means that in each iteration, only one row of the distance matrix is computed and stored in an in-place manner. Some works use alternative computation and update schemes where, in each iteration, one row or one diagonal is computed.

See Figure 2.11 for a better illustration of this scheme. This figure shows the distance matrix and computation associated with a **self-join** problem where the distance matrix is symmetric and computing half of the elements of the distance matrix is sufficient. Also,



**Figure 2.11:** Illustration of **STOMP** method to compute the matrix profile. We highlight the relation between the distance matrix and the matrix profile and the corresponding elements updated in each iteration (highlighted in green and blue).

note that in the case of **self-join** problems, an exclusion zone (which was introduced and discussed in Section 2.3.1) needs to be considered, and its corresponding distance computations need to be ignored. This eliminates the need to compute the elements of the matrix around the diagonals of the distance matrix that are colored in pink.

In Figure 2.11, the row computation corresponding to a running iteration is highlighted in green. **STOMP** scheme stores and updates a vector of distances associated with a single row of the distance matrix in an in-place fashion. This vector is not illustrated in the image, but one can imagine it as a vector storing the elements in green which represent the target elements of the distance matrix that are computed in each iteration. The computed distances in each iteration (green elements) are then compared against the best-so-far computed matrix profile and matrix profile index to update them accordingly (elements colored in blue).

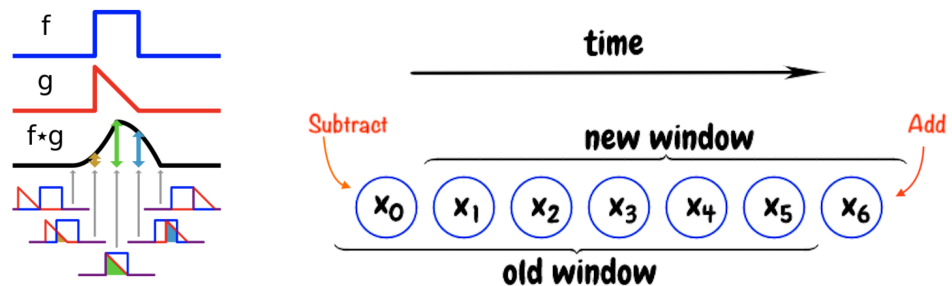
So far, we have described **STOMP** as an iterative in-place method for matrix profile computation, which improves memory utilization compared to the naive approach. However, **STOMP** is more powerful than only that. For cost-efficient computation, **STOMP** relies on a variation of the so-called *streaming dot product* formulation [ZKS<sup>+</sup>19] that

## 2 Background and Technical Foundations

is introduced in Table 2.3. At the core, this approach relates the computation of the elements of the distance matrix  $D$  between two subsequences to the distances between subsequences in their immediate neighborhood (see this relation in Equation 2.7). This formulation reduces the computational costs for calculating the distance between two subsequences by a factor of  $m$  (i.e., the subsequence length), as discussed in Section 2.3.4.

Before digging deeper into the details of streaming dot product formulation, we further clarify the rationale behind it by looking at the pieces of the term “streaming dot product”:

1. dot product: in signal processing, often cross-correlation of signals is used as a means to measure similarity. Often, cross-correction factors can be formulated based on computing a dot product between the signals. For example, the Pearson’s Correlation (PC) factor [Kir08] can be computed from the dot product between the input signals (see Equation 2.6). Also, commonly, the correlation factors (similarity measures) in such analyses can be related to distance measures, such as Euclidean distance (see Equation 2.7). Therefore, for matrix profile computation, there is a natural path to compute the similarity between a pair of subsequences based on the dot product between them. Streaming dot product formulation does exactly that, but in a highly efficient manner (Equation 2.8).
2. streaming: again, in signal processing, we often deal with sliding windows. In such cases, streaming formulations are often highly efficient. A simple example is the computation of moving (sliding) average illustrated in Figure 2.12. In this case, when the average of values associated with the first (old) window is given, the average for the new window can be computed by only two operations: by first eliminating the contribution of the first element of the old window to the average ( $x_0$  in Figure 2.12), and next adding the contribution of the last element of the new window ( $x_6$  in Figure 2.12). This is much cheaper compared to the naive computation of the average for the new window, which requires 5 operations in the example of Figure 2.12 (i.e.,  $\mathcal{O}(m)$ ). Streaming dot product applies the same sort of enhancement for computing the dot products (Equation 2.8).



**Figure 2.12:** Illustration of cross-correlation of signals (left) [Dau23] and streaming formulation for moving average (right) [Nes23].

We discuss more details by showing a variation of streaming dot product-based in Table 2.3 to address [self-join](#) matrix profile computation. This formulation relies on a streaming dot product to compute Pearson’s correlation between subsequences and transform this correlation to [z-normalized](#) Euclidean distance.

The streaming dot product works as follows: first, the vectors  $\mu$ ,  $d^{-1}$ ,  $df$ , and  $dg$  (see Equations 2.1- 2.4) for the input time series are computed. This is a cheap computation and can be done in  $O(n)$ . These vectors, in essence, represent the means, inverse of norm-1, and difference of the first and last sample in subsequences, and the sum of deviations from the mean in the first and last samples in each subsequence. We skip describing the semantics of these vectors, but they are used to compute  $\overline{QT}$  (Equation 2.5), which is a form of dot product between the subsequences of the input series. The values of  $\overline{QT}$  can be easily translated to Pearson’s correlation factors using Equation 2.6 as well as to the [z-normalized](#) Euclidean distances using Equation 2.7.

We put the equations presented in Table 2.3 into a Pseudocode 2 to show how they are used to compute the matrix profile in a [self-join](#) setting. Under this setting, the distance matrix is symmetric, and computing the upper triangular part of it is sufficient (see Figure 2.11).

---

Average of samples in the subsequence starting at record index $i$	$\mu_{i+1} = \mu_i + (T_{i+m} - T_i) / m$	(2.1)
Inverse of norm-1 of samples in a subsequence starting at record $i$	$d_i^{-1} = \left( \sum_{z=0}^m T_{i+z} - \mu_i \right)^{-1}$	(2.2)
Intermediate values used in mean-centered streaming dot product formulation 2.5	$df_{i+1} = (T_{i+m} - T_i) / 2$	(2.3)
Intermediate values used in mean-centered streaming dot product formulation 2.5	$dg_{i+1} = (T_{i+m} - \mu_{i+1}) + (T_i - \mu_i)$	(2.4)
Streaming dot product of samples in subsequences $i$ and $j$ [ZKS <sup>+</sup> 19]	$\overline{QT}_{i+1,j+1} = \overline{QT}_{i,j} + df_i \times dg_j + df_j \times dg_i$	(2.5)
Pearson correlation among subsequences starting at records $i$ and $j$	$\rho_{i,j} = \overline{QT}_{i,j} \times d_i^{-1} \times d_j^{-1}$	(2.6)
Euclidean distance among subsequences starting at records $i$ and $j$	$D_{i,j} = \sqrt{2 \times m \times (1 - \rho_{i,j})}$	(2.7)
$i$ -th distance profile	$\delta_i = D_{i,*}$	(2.8)
Matrix profile and its index	$P_i = \min(\delta'_i), \quad I_i = \operatorname{argmin}(\delta'_i)$	(2.9)

---

**Table 2.3:** Streaming dot product formulation in [STOMP](#).

## 2 Background and Technical Foundations

Here are the steps to compute the matrix profile:

1. The first step is to (pre-)compute vectors  $\mu$ ,  $d^{-1}$ ,  $df$ , and  $dg$ , using Equations 2.1–2.4 in Line 1 of Pseudocode 2.
2. In Line 2, Pseudocode 2, a streaming dot product, is initialized with the distance between the *first* subsequence in the time series and all other subsequences. This initialization is done by a brute force computation of the dot product (similar to Line 3 in Pseudocode 1)
3. Next, the upper triangular part (Lines 3–4, Pseudocode 2) of the distance matrix (except for the exclusion region  $e$ , i.e., a region in Figure 2.11 highlighted in pink) is computed and is used to iteratively compute the matrix profile and index (Lines 3–10, Pseudocode 2):
  - The Equations 2.5 is employed to iteratively—and in-place—update the dot product vector (Line 5, Pseudocode 2).
  - Next, the computed dot product values are converted to Euclidean distance ( $\delta$ —Lines 7, Pseudocode 2).
  - Finally, the computed distance vector is used to update the matrix profile and index values. This is done by a) finding the minimum value in the distance vector  $\delta$  and merging it to the results (Lines 8, Pseudocode 2), and b) merging all values of  $\delta$  to the resulting matrix profile and index by using element-wise minimum (Lines 9, Pseudocode 2. Again visit Figure 2.11.).

---

**Pseudocode 2** Procedure for computing the multi-dimensional matrix profile based on **STOMP**.

---

**Input:** d-dim. time series  $T^r \in \mathbb{R}^{nr}$ ,  $T^q \in \mathbb{R}^{nq}$  and subsequence length  $m \in \mathbb{N}$ .

**Output:** Matrix profile  $P \in \mathbb{R}^{n-m+1}$  and matrix profile index  $I \in \mathbb{Z}^{n-m+1}$ .

---

```

1:  $\mu, d^{-1}, df, dg \leftarrow \text{precompute\_statistics}(T^r, m)$  ▷Equations 2.1– 2.4
2:  $QT \leftarrow \text{initialize\_streaming\_dot\_product}()$  ▷see Pseudocode 1
3: for  $i \leftarrow 1$  to  $n - m$  do
4:   for  $j \leftarrow i + e$  to  $n - m$  do
5:      $QT[j] \leftarrow QT[j - 1] + df[i - 1] * dg[j - 1] + df[j - 1] * dg[i - 1]$  ▷see Figure 2.11
6:   end for
7:    $\delta \leftarrow \text{convert\_to\_euclidean\_distance}(QT)$ 
8:    $P[i], I[i] \leftarrow \text{minimum}(\delta)$ 
9:    $P, I \leftarrow \text{element\_wise\_min}(P, \delta)$  ▷vector updates
10: end for

```

---



## 2.4 Background on Computer Architecture and High Performance Computing

To better understand the methods and experiments discussed in this work, we first provide a general overview of computer architecture, parallelization as a core concept, and high-performance computing. We mainly adopt and discuss the concepts from the textbook on computer architecture by Hennessy and Patterson [HP11] and dissertation with similar backgrounds [Yan20, Eng21, Net22, Mai21].

The Von Neumann architecture [vN45] is a fundamental concept in computer architecture that describes a computer design consisting of a memory system that holds both program instructions and data, which can be fetched and operated on by a central processing unit. This architecture was introduced by mathematician and computer scientist John von Neumann in the 1940s, and while modern computer systems rely on the Von Neumann’s architecture in many aspects, they introduce architectural enhancements to improve performance and efficiency, and therefore, they deviate from it, e.g., by incorporating caching, pipelining, and parallelism. This applies to both CPU and GPU platforms. Historically, various architectural innovations, including a boost in clock speed, execution optimization, and caching, contributed to the development and progress of computer systems. However, since 2005, parallelization has played an important role as the “free performance lunch” provided by chip manufacturers boosting clock speeds in mainstream systems started to fade due to reaching the hard physical limits [Sut13]. Since then, various parallelization technologies, including hyper-threading, multi-core, and multi-threading, have been the driver of the performance in computer systems.

These technologies, in essence, rely on four types of parallelism in hardware [HP11, Yan20]:

1. Instruction-Level Parallelism, corresponding to techniques to increase instruction throughput within a single cycle and includes concepts like pipelining and superscalarity and Very Long Instruction Word (VLIW). Consider a program as an instruction stream where each instruction can be split into multiple stages on modern processors: instruction fetch **IF**, instruction decode **ID**, execution **EXE**, memory access (**MEM**), and writeback (**WB**). With pipelining, the executions of multiple instructions are overlapped to hide instruction latencies.
2. Vector Parallelism, which is a type of parallelism often abundantly used in Vector Architectures (such as accelerators and vector machines) and Graphic Processor Units (**GPUs**). Often, in classifications like this, vector parallelism is considered part of instruction-level parallelism. However, we are elevating it to a separate class as it is often more explicitly exposed to the compilers and applications. Vector parallelism relies on the Single Instruction applied to Multiple Data entries (**SIMD**) loaded into a vector register. The concept originated from the vector processors developed in the 1970s (STAR-100 [Sch87] and Cray-1 [Rus78]), allowing for processing vector instructions. Modern processors are equipped with special vector units for enabling data parallelism. Examples are Advanced Vector Exten-

## 2 Background and Technical Foundations

sions with 512-bit wide vector registers and instructions ([AVX512](#)) [[Rei13](#)] used initially in Intel's Xeon Phi (Knights Landing) and later in Intel's X86 processors (and recently [EPYC](#) 4th generation server processors). Other examples of [SIMD](#) are the Advanced [SIMD](#) Extension ([NEON](#)) and Scalable Vector Extension ([SVE](#)) in the Arm architecture featuring variable-length vector registers from 128 to 2048 bits wide used in Fujitsu's [A64FX](#) processor. [GPUs](#) also exploit this type of parallelism.

3. Thread-Level Parallelism, where each thread executes an independent instruction stream. Threads can run in parallel on different physical [CPU](#) cores of a multi-core processor or hyper-threads of a [CPU](#) core exposed by Simultaneous Multithreading ([SMT](#)). [GPUs](#) also heavily rely on thread-level parallelism: the threading model in [GPUs](#) is based on Single Instruction Multiple Thread ([SIMT](#)), where groups of threads in the so-called thread block execute the same instruction stream, but each apply it on a different data element.
4. Process-Level Parallelism exploits parallelism among multiple compute resources typically at a coarse level (e.g., among multiple machines). This level of parallelism is often enabled by the mechanisms provided in operating systems, e.g., by running multiple Linux processes. This includes multi-processing on a single (or multi-socket) multicore system, multi-processing on multi-socket processors that share the memory subsystem, or multiple (potentially even) physically disjoint computers (nodes). We can also include parallelism among multiple accelerators ([GPUs](#)) accelerators that are commonly sitting the nodes and are connected via [PCIe](#) (Peripheral Component Interconnect Express) or, in recent designs, connected via [CXL](#) (Compute Express Link) and are driven by multi-processing schemes in this category.

Application Programmers (e.g., in [HPC](#)) typically are exposed to the parallelism in hardware in two parallelization patterns:

- Data-Level Parallelism: this parallelization appears in applications where many data items can be operated on in parallel.
- Task-Level Parallelism: in this parallelization pattern, the problem at hand can construct a sufficiently large of (ideally) independent tasks that can be operated on in parallel.

These patterns in applications and parallelism in hardware, and in general parallel computing, were classified by Michael Flynn [[Fly](#)] based on the parallelism in the instruction and data streams. This classification scheme is called [Flynn's Taxonomy](#): This taxonomy defines four categories: [SISD](#) (Single Instruction stream, Single Data stream), [SIMD](#) (Single Instruction stream, Multiple Data streams), [MISD](#) (Multiple Instruction streams, Single Data stream), and [MIMD](#) (Multiple Instruction streams, Multiple Data streams). In [SISD](#) architectures, a single instruction stream operates on a single data

stream, typical of traditional sequential computers. However, instruction-level parallelism techniques such as superscalar and speculative execution can be employed. **SIMD** architectures use a single instruction stream to process multiple data streams simultaneously, often seen in vector processors, multimedia extensions to ISAs, or **GPUs** to facilitate vectorization parallelism. **MISD** architectures involve multiple instruction streams operating on a single data stream concurrently; however, they are very uncommon, with mainly fault tolerance use cases: the Space Shuttle flight control computer [Yan20] is worth mentioning here. Finally, **MIMD** architectures support multiple instruction streams operating on multiple data streams concurrently, as found in most parallel computers and clusters. Each processing element fetches its own instructions and operates on its own data. **MIMD** is flexible and can cover various Data/Thread/Process level parallelism.

### 2.4.1 Architecture of Modern and Emerging HPC Systems

Modern top-tier High-Performance Computing (**HPC**) systems are designed and optimized for processing large and complex computational tasks, including simulations in physics, material science, astronomy, energy, climate, weather and environmental simulations, nuclear and fusion, as well as biomedical research—recently, the deployment of **AI** use cases in **HPC** has also gained a large momentum. These systems are large clusters of individual compute nodes tightly coupled together to a high-performance storage layer through a high-performance network fabric. These nodes work together to execute complex calculations in parallel. These systems are built to deliver high computational power in terms of **flop/s** rates, high memory bandwidth, and massive throughput, as well as low latency communication between components, allowing the application to scale horizontally on top of them.

Each compute node typically includes a high-end data-center class server (i.e., a high-end multi-core processor) designed for parallel processing and is potentially equipped with one or many accelerators (e.g., **GPUs**). In **CPU**-based **HPC** systems, the bulk of performance is delivered by multiple multi-core processors, while in accelerated-based **HPC** systems, the accelerator brings in the bulk of performance. The interconnect fabric is a key component in **HPC** systems as it glues all the components and enables fast communication between them. It allows message passing and synchronization between nodes, as well as fast **I/O** to the storage nodes. Modern **HPC** systems often employ high-speed interconnect technologies like InfiniBand and **Omni-Path** with Remote Direct Memory Access (**RDMA**) capabilities for low-latency, high-bandwidth communication.

On the storage side, **HPC** systems require high-capacity and high-performance storage solutions to handle large datasets and provide low-latency and high bandwidth parallel access to data during computation. Parallel File Systems (**PFS**) like Lustre and Spectrum Scale (**GPFS**) are commonly used to spread data across multiple storage backbones, enabling parallel access from multiple compute nodes [SKJJ16, Bra05, IBM07].

**HPC** clusters consume a significant amount of power, in order of one to tens of Mega Watt, and produce a significant amount of heat due to the density of computing com-

## 2 Background and Technical Foundations

ponents. Direct liquid cooling mechanisms are increasingly employed and are crucial to prevent overheating of hardware components and chips and to ensure stable operation.

HPC clusters are typically shared among multiple users running various jobs simultaneously. Users interact with the systems through job scheduling resource management by providing computational resource allocations, e.g., with the granularity of nodes. Users need to employ specialized software stacks to exploit the parallelism and performance features of the hardware. This includes parallel programming models and libraries, like the Message Passing Interface (MPI) [WD96] for distributed memory programming, OpenMP [DM98, Ope08, CDK<sup>+</sup>01] for shared memory (primarily), and CUDA [NVF20] and HIP (for GPU acceleration), as well as optimized numerical libraries, such as BLAS [BDD<sup>+</sup>02, vdGG11] and LAPACK [ABB<sup>+</sup>99], and more high-level performance libraries such as PETSc [BGMS98]. Additionally, proper use of tools [ABF<sup>+</sup>10, BGB<sup>+</sup>16, SGM<sup>+</sup>08], compilers and code generation techniques [LA04, Eng21], runtime libraries, and debugging utilities [NS07] are essential for the development and efficient execution of applications on HPC systems.

Despite the recent trends in the shift of architectures of HPC systems towards extensive use of accelerators, CPU-based systems are abundant. Three out of the top ten in the TOP500 list are CPU-based systems. A worthy mention is the Fugaku supercomputer installed at RIKEN Center for Computational Science in Japan in 2020, which uses Fujitsu A64FX Arm-based processors. This system is highly praised by the community for paving the path to the exascale era, as well as serving as the number one system for four TOP500 rounds and the number two on the list to this day.

Specifically relevant to this work is the SuperMUC-NG installed by Lenovo at the Leibniz Supercomputing Centre in Munich, Germany, in 2018. This system appeared in place number eight in TOP500 with 19.48 petaflop/s  $R_{\max}$  performance. This system is currently ranked 31 worldwide and is the second fastest HPC system in Germany. SuperMUC-NG [Bay20] comprises 6,336 compute nodes with an aggregated main memory capacity of 719 TB and a peak performance of 26.9 petaflop/s. SuperMUC-NG has a homogeneous architecture in which all the compute nodes include two Intel Skylake Xeon Platinum 8174 processors. The interconnect is an Omni-Path network with 100 Gbit/s bandwidth, with a fat tree network topology consisting of eight islands connected with a blocking factor (pruning factor) of 1:4.

SuperMUC-NG consumes around 3 Mega Watt and uses direct warm water cooling. It uses IBM Spectrum Scale [IBM07] as its Parallel File System with 50 petabytes capacity and 500 GB/s aggregated bandwidth. SuperMUC-NG runs SUSE Linux (SLES) and SLURM [YJG03] job scheduler. See the link to the specification of SuperMUC-NG in Appendix A5, where more details about this system are covered.

At the same time, considering the trends in the extensive use of accelerators and, specifically, GPUs in HPC, it is very important to cover the type of HPC systems that use accelerators. In these systems, compute nodes are often equipped with multiple GPUs that are potentially connected to each other through fast communication buses (NVIDIA NVLink or AMD Infinity Fabric) to enable high-bandwidth and low-latency communication. The bulk of performance is delivered by the GPUs. Two out of the top

## 2.4 Background on Computer Architecture and High Performance Computing

three systems in the **TOP500** list are GPU-based: **Frontier**, which is the first **exascale** system installed in **US DOE Oak Ridge National Laboratory** by **HPE**, has nodes consisting of one AMD 3rd Generation **EPYC** processor, and four **AMD Instinct MI250X GPUs**. **Frontier** uses a Slingshot 11 network fabric in a three-hop dragonfly topology to connect the compute nodes.

In particular noteworthy for the work presented in this thesis are the **Raven-GPU** system at the Max Planck Computing and Data Facility (with four **NVIDIA A100-SXM4** per node) and the **Selene** supercomputer at **NVIDIA Corporation** (with 8 **GPUs** per node and **NVLink**), including **NVIDIA V100** and **NVIDIA A100 GPUs**, which are the dominant data center **GPUs** in the market at the time of development of this work.

Emerging **HPC** systems employ more heterogeneity and comprise special-purpose components and accelerators such as **GPUs** and **SmartNIC** in addition to general-purpose **CPUs**. Modular supercomputers [**Gmb21a**, **Gmb21b**] and the integration of quantum accelerators [**MSR22**, **RTL<sup>+</sup>22**] and **AI** accelerators [**Lav22**] are also new architectural trends. Also, another influential trend that is impacting **HPC** architecture is the appearance of desegregated memory [**GKB<sup>+</sup>23**].

Also architectures to enable domain-specific hardware for time series analysis, e.g., **FPGAs**, and special purpose hardware such as **AI** accelerators like **TPUs** [**JYP<sup>+</sup>17**], **Graphcore's IPU** [**Res22**] and **Cerebras WaferScale Engine** [**Lav22**] are also interesting for investigation of time series mining workloads

### 2.4.2 Scalability in HPC

A core concept in the acceleration of application on **HPC** systems is leveraging the horizontal scaling of the resources available to the application. With horizontal scaling of resources, two main scaling paradigms for **HPC** applications are often considered. Scaling by throwing more resources to work on the set of tasks associated with a problem of a fixed size in parallel. This is known as strong scaling. However, usually, performance improvement, e.g., the latency of execution or throughput of processing, does not scale linearly with the number of resources that collaborate to solve the problem. Often, communication and synchronization among the processing elements are required for the correct execution of an application. This imposes performance overheads that are not present in a sequential execution. Related to strong scaling, Amdahl's law [**Amd67**, **CSG98**] is a well-known principle for calculating the maximum theoretical speedup for the parallel execution of a given task (e.g., a problem with a fixed size) and the potential speedup based on the portion of the task that can be parallelized. Amdahl's law states that the overall speedup,  $s$ , that can be achieved by parallelizing a computational task is limited by the sequential fraction,  $0 \leq f \leq 1$ , of the task that cannot be accelerated through parallelization (e.g., with  $p$  processes). The law mathematically reads:

$$s(p) = 1/[f + (1 - f)/p] \quad (2.10)$$

The law implies that, as the number of processors increases, the potential speedup of a computation task becomes limited by the sequential portion related to the so-called

critical path of the task, which is sequential and cannot be parallelized. In other words, no matter how many processing elements are used, there will always be a maximum speedup that can be achieved due to the nature of certain parts of the task that cannot benefit from parallelization. Therefore, Amdahl's law provides a simple model for parallel performance optimization and potential performance gains.

The second scaling paradigm for HPC applications is called weak scaling. While strong scaling focuses on fixed problem sizes, weak scaling considers scaling the problem size proportional to the increase in compute resources.

Gustafson's Law [Gus88] is the reincarnation of Amdahl's law to the weak scaling perspective. Gustafson's Law describes how the performance of applications improves when scaling the problem size proportional to the computational resources. It also shows that the fraction of non-parallel parts,  $0 \leq f \leq 1$ , reduces linear scalability by a factor of  $1 - f$ . The law mathematically reads:

$$s(p) = f + (1 - f)p \tag{2.11}$$

### 2.4.3 Performance Optimization in HPC

Running applications on large HPC systems typically involves interacting with various subsystems, including compute, network, and storage components. Performance optimization in HPC targets coordinating the resources and efficient use of the components. HPC applications are designed and optimized for efficient execution at the node level as well as at scale. Therefore, various aspects of optimization contribute to overall application performance.

At the node level, various aspects and challenges, including memory access patterns, data locality, caching, parallelism and vectorization, node-topology awareness, and affinity and load balancing, are often investigated and addressed. Additionally, reduced- and mixed-precision computation methods are drawing increasing attention. These methods often intend to leverage the hardware support for computation with fewer bits and can significantly impact various aspects of application performance, including memory footprint and cache utilization, and boost the flop/s rates. Overall, low-precision computation allows for better and more efficient use of hardware resources. However, they often come at the price of reduced accuracy. Therefore, an important aspect of employing such methods is to understand the overall impact on the accuracy and performance of applications. In other words, the main challenge here is to quantify the performance-accuracy trade-offs.

On the system level, typically after algorithmic considerations, various optimization and tuning techniques are employed, including the use of non-blocking communication and overlapping communication and computation, topology-aware communication, and collective and layout optimization, as well as enabling RDMA. Load balancing is also an important optimization aspect at large scale. Overall, the target for investigation of such optimization techniques is to maximize network bandwidth utilization as well as minimize synchronization and communication latency.



On the storage side, the use of parallel I/O, e.g., to access single or multiple files in parallel, object storage, and data compression are often explored, with the intention to exploit high bandwidth to storage and minimize disk utilization.

In addition to the above-discussed optimizations, algorithmic redesign and improvements play an important role in HPC. Such algorithmic optimizations can include various aspects of numerical schemes, spacial and time integrations, and communication-avoiding algorithms, as well as exploiting acceleration data structures, e.g., redesigning algorithms to exploit hashing mechanisms or tree data structures to improve search and sort operation within the core of existing algorithms.

### 2.4.4 Performance Optimization Tools in HPC

The process of optimizing applications in HPC at various levels often involves using profiling, tracing, and instrumentation tools to acquire detailed data about application performance, bottlenecks, and overheads. This data is combined with performance models (roofline [WWP09] or LogP [CKP<sup>+</sup>93]), as well as performance visualization tools to provide insights to the application developers. Application developers use the insights to improve the performance of applications iteratively. While this process is often manual, HPC sites try to incorporate systematic application performance monitoring, instrumentation, and introspection to support production runs on HPC systems with minimal and acceptable performance penalties; such approaches can also assist developers by providing key performance metrics automatically.

At the node level, modern CPUs are equipped with performance monitoring units (PMUs) that provide hardware performance counters for measuring hardware events such as cache misses and branch mispredictions. Tools such as GNU perf, PAPI [BDG<sup>+</sup>00, TJYD09], and Likwid [THW10] can facilitate accessing these counters. In addition, on Intel CPUs, tools such as Intel VTune provide similar statistics combined with visualization and some hints on optimizations. Similar tools for GPUs exist (e.g., NVIDIA Nsight Compute), which can provide statistics about the activities of running GPU kernels. On the system level, various tools, including Profiling Tool for MPI (MPIP) [VC06] for lightweight profiling of communication, Intel Trace Analyzer, and Collector (ITAC) for message tracing, are often used. Score-P is a code instrumentation and run-time measurement framework supporting communication tracing. Scalasca [KRM<sup>+</sup>12], TAU [SMM06], and HPCToolkit [ABF<sup>+</sup>10] offering similar tracing, and profiling capabilities for parallel applications.

Manual source code instrumentation (with user-defined markers as well as marker APIs in PAPI [BDG<sup>+</sup>00, TJYD09], LIKWID [THW10], and Caliper [BGB<sup>+</sup>16]) helps developers to instrument specific regions/sections in the code, e.g., to read counter values associated only with the specific regions of the code. In this work, we heavily rely on this type of instrumentation to get insights about hotspots, bottlenecks, and overheads.

### 2.4.5 Programming Models in HPC

On node level, in addition to vendor-specific libraries, e.g., [CUDA](#) [NVF20], [HIP](#) [Adv], and [SYCL](#) [KRH15], [OpenMP](#) [DM98, Ope08, CDK<sup>+</sup>01] plays an important role in [HPC](#) applications as a portable standardized model for programming multi-core systems and accelerators. [Message Passing \(MPI\)](#) [WD96], [Partitioned Global Address Space \(PGAS\)](#) [Alm], e.g., [GASPI](#) [ABB<sup>+</sup>13], and [OpenSHMEM](#) [CCP<sup>+</sup>10], and [Task-based models](#) (e.g., [Charm++](#) [KK93]) are well-established programming paradigms in [HPC](#). Among these models, [MPI](#) is the defacto model that is widely used in classical [HPC](#) codes.

[MPI](#) is a standardized communication protocol and a prominent programming model in [HPC](#). [MPI](#) was introduced in the 90s, and, since then, the [MPI Forum](#) has been responsible for its standardization.

[MPI](#) follows the [MPMD \(Multiple Program Multiple Data\)](#) model (close to [MIMD](#) in [Flynn's Taxonomy](#)). To facilitate message passing and [MPMD](#) paradigms, [MPI](#) specifies a number of concepts and abstractions, including process, group, and communicator. Although [MPI](#) can be used in [MPMD \(Multiple Program Multiple Data\)](#) modes, it is commonly used for [Single Program Multiple Data \(SPMD\)](#), where execution of a single binary is spawned by an [MPI launcher](#) as multiple [MPI Processes](#) (which are often operating system processes) on single or multiple compute nodes. [MPI](#) defines a group of processes and communication context (communicators) for message passing among the members of the group. Each [MPI](#) process acquires a unique identifier within a communicator, which is called the rank. Ranks are used by the application as IDs, e.g., in workload distribution, and partitioning. Through these functionalities, [MPI](#) facilitates efficient and scalable communication on high performance interconnects, such as Intel [Omni-Path](#) and [NVIDIA Infiniband](#) networks. [MPI](#) standard specifies a wide range of functionalities for message passing. Specifically, it defines syntax and semantics for functions in [C](#), and [FORTRAN](#). These include so-called peer-to-peer ([P2P](#)) communication, e.g., send and receive between a pair of processes, as well as so-called collective communication among a set of processes, e.g., reductions.

[MPI](#) also defines functionalities to enable non-blocking communication modes, where processes can initiate a communication (e.g., [P2P](#), collectives ...) without blocking the control for the completion of the communication. Non-blocking communication allows for overlapping of communication and computation, thereby potentially improving the performance and scalability of parallel applications. Exploiting non-blocking [MPI](#) operations in [HPC](#) applications is also a well-established technique to enable the overlapping of communication and computation phases of applications, offering better scalability.

In this work, we rely on using such non-blocking operations to improve the scalability of time series mining.

### 2.4.6 GPUs in HPC and Reduced- and Mixed-Precision Computation

[GPUs](#) are specialized processors designed originally for processing graphic rendering. Due to the inherent needs of graphic processing (e.g., operating on many pixels in par-



allel), these devices are designed for large amounts of parallelism among many cores. GPUs evolved beyond this initial purpose through significant academic contributions in the late 1990s and early 2000s. Pioneers like Hanrahan at Stanford University influenced their use in general-purpose computing, as evidenced by Ian Buck’s Brook project in 2001 and Mark Harris’s research in 2003, both focusing on employing GPUs for non-graphic parallel computing tasks [BFH<sup>+</sup>04]. The concept of running a general computation on GPUs was brought to the mainstream by NVIDIA Corporation with the introduction of their CUDA (Compute Unified Device Architecture) platform [NVF20].

On the architectural level, unlike CPU cores which are optimized for minimal latency, GPU cores are designed for high throughput. Additionally, compared to modern CPUs, GPUs benefit from high bandwidth memory (GDDR), which can bring performance advantages to HPC applications. Also, GPU cores are typically running at lower frequencies compared to CPU cores, resulting in relatively high energy efficiency for running many HPC and AI applications.

GPUs employ SIMT (Single Instruction Multiple Thread) as the execution model, in which groups of threads, called thread blocks, execute the same instruction on various data entries. GPUs include so-called streaming multiprocessors (SMs<sup>2</sup>), each consisting of processing units called GPU cores that are capable of arithmetic and logic operations called. Through a so-called warp scheduler, SMs execute collections of threads called warps in lockstep, allowing for per-SM instruction-level parallelism and maximizing temporal and spacial locality with respect to the instruction and data pipelines.

NVIDIA GPUs also include tensor cores that support reduced-precision arithmetics, i.e., FP32, FP16, and TF32. Additionally, these GPUs support a concept called CUDA streams to declare and later trigger a sequence of operations that will be executed in issue-order on the GPU. In many cases, HPC applications require the incorporation of tiling or batching schemes for data locality reasons or lack of enough GPU memory. In such cases, CUDA streams are highly beneficial to enable a coarse-grained pipeline and tile scheduling schemes for moving data and launching GPU kernels.

## 2.5 Scope of this Work

Time series mining, especially similarity indexing and matrix profile computation, requires an intensive computational effort and often large data volumes (see Sections 2.2.1 and Section 2.3.4). Leveraging the capabilities of HPC systems, including the abundance of compute resources, with high memory capacity and bandwidth as well as high speed interconnects, are highly relevant to data mining, particularly time series mining, allowing for faster, more efficient analysis (see Section 2.4.1). Additionally, HPC systems allow for running parallel time series mining algorithms at large scale with high efficiency by providing the high-speed interconnect and overall reducing the time to solution (see Section 2.4.3). In this thesis, we investigate approaches and techniques that are

---

<sup>2</sup>We are following NVIDIA terminology. Compute Unit (CU) is often used in the context of AMD’s GPU architecture

## 2 Background and Technical Foundations

overall related to the optimization techniques discussed in Section 2.4.3, in particular, node-level, system-wide, and algorithmic optimizations.

Additionally, in an orthogonal direction, investigating the capabilities of GPUs, their massive parallelization, and their built-in reduced-precision capabilities (as discussed in Section 2.4.6) in the context of time series mining is another interesting research avenue. In the context of this work, we are mainly focusing on NVIDIA data center GPUs with the Tesla architecture, which are commonly used in HPC. While Intel GPUs and AMD GPUs, as well as other AI accelerators, are interesting to investigate, we limit the scope of evaluations to NVIDIA GPUs. However, the concepts presented in this work can also be applied to the GPUs of other vendors. Of all data formats discussed in Section 2.1.6, this thesis considers FP64, F32, and FP16 relevant for reduced- and mixed-precision investigations as they are typically supported in mainstream HPC hardware. While all the other formats, especially those specific to machine learning, are also interesting to investigate, we consider them out of the scope of the studies in this thesis.

Overall, approaches intend to leverage the advantages of HPC systems for time series mining. For that, we highly rely on the mainstream HPC environments discussed in Section 2.4.5 and the various tools for performance debugging, optimization, and tuning that are discussed in Section 2.4.4. Finally, This work focuses on today’s dominant architectures based on GPUs or general-purpose CPUs, and the investigation of the emerging system designs and architectures discussed in Section 2.4.1 fall out of the scope of this work.

### 2.6 Summary

We started this chapter by providing a brief background on time series, multi-dimensional time series, and their definitions and concepts like time series windows and subsequences that are essential for the rest of this thesis. We introduced various formats and storage layouts. We then provided illustrative examples, which we will revisit in the next chapter to apply the methods introduced in this thesis. We introduced the concept of matrix profile and described the foundation of matrix profile computation methods.

Additionally, we covered various concepts from computer architecture and HPC that are essential for understanding the topics in the rest of this thesis. Specifically, we provided an overview of the architecture of modern CPU- and GPU-based HPC systems. We covered various fundamental aspects of scalability, optimization techniques, programming models, and tools in HPC. Finally, we briefly discussed the fundamentals of reduced- and mixed-precision computation in HPC.

## 3 Overview of the Work in the Literature

In this chapter, we provide a brief overview of the relevant existing literature and research. Specifically, we look into the following topic: we cover the general methods for scalable time series mining, high performance computing, and approaches for efficient large-scale data mining. We look at the brief history of matrix profile emergence and review the methods and approaches to compute the matrix profile for single and multi-dimensional time series. Finally, we look at the exact and approximate methods for the computation of the matrix profile.

### 3.1 Scalable Methods for Time Series Mining

There is extensive literature on mining time series, including several survey papers [RS02, Fu11, EA12]. In particular, in the context of scalable and offline processing of time series data, there are a number of studies on scalable solutions in the literature to review: Huang et al. [HZM<sup>+</sup>16] target the use of Apache Spark for speeding discord discovery in time series. Moreover, the conducted experiments in this work are limited to 10 nodes. Berard et al. [BH13] scale time series similarity search to 20 nodes on a Hadoop cluster. Sart et al. [SMN<sup>+</sup>10] and Zhu et al. [ZZS<sup>+</sup>18] use accelerators for speeding time series mining workloads. Movchan et al. [MZ15] study time series similarity search on the Intel Many-core Accelerators, utilizing OpenMP. However, none of the above studies address mining on large-scale HPC systems, and analyses are very limited to specific targets of similarity search. Overall, there is a large gap in the investigation of scalable time series mining methods, especially on HPC systems.

### 3.2 History of Matrix Profile

Matrix profile was first introduced by Yeh et al. [YZU<sup>+</sup>16] as a new general approach that is capable of generating comprehensive similarity annotations for time series indexing in order to simplify many data mining tasks. This approach generates annotation vectors that are encoded with the similarity of windows of time series with a high density of similarity information. The similarity index generated by matrix profile has implications for many, perhaps most, time series data mining tasks [Keo23]. Matrix profile computation can benefit from the use of inertial properties of time series, e.g., the similarity of neighboring windows, for efficient computation of similarity annotations. Also, the computation of matrix profile exposes a high degree of parallelism, allowing for the use of parallelism in modern processors and accelerators. For these reasons, the matrix

### 3 Overview of the Work in the Literature

		Method	Accuracy	Model	Environment
offline	single-dim	STAMP [YZU <sup>+</sup> 16]	Exact	Matlab	CPU
		STOMP [ZZS <sup>+</sup> 18]	Exact	CUDA	GPU
		SCRIMP++ [ZYZ <sup>+</sup> 18]	Both	C++	CPU
		ScrimpCo [RVR <sup>+</sup> 20]	Exact	OpenCL	CPU&GPU
		SCAMP [ZKS <sup>+</sup> 19]	Both	Google Cloud	GPU
		MP-MPI [Pfe19]	Exact	HPC	CPU
online	single-dim	LAMP [SZSF <sup>+</sup> 19]	Approx.	Keras	CPU
		FA-LAMP [KZB22]	Approx.	HLS	FPGA
		DAMP [LWM <sup>+</sup> 22]	Exact	Matlab	CPU
offline	multi-dim	mSTAMP [YKK17]	Exact	Matlab	CPU

**Table 3.1:** Overview of methods to compute the matrix profile.

profile has gained significant momentum in the data mining community, and various approaches for efficient computation of the matrix profile have been proposed and studied. These approaches and their characteristics are listed in Table 3.1.

### 3.3 Methods to Compute the Matrix Profile

Yeh et al. [YZU<sup>+</sup>16] proposed the STAMP algorithm that uses a series of FFT operations to compute the distances needed for matrix profile computation. This algorithm takes  $O(n^2 \log n)$  time to compute for a time series of length  $n$  (assuming self-join matrix profile under Euclidean distance). The paper also introduced the Scalable Time series Anytime Matrix Profile (STAMP) method to compute the matrix profile. Following this approach, various algorithms have been proposed to improve the efficiency of the computation, including Scalable Time series Ordered Matrix Profile STOMP [ZZS<sup>+</sup>18] and Time Series Motif Discovery at Interactive Speed (SCRIMP++) [ZYZ<sup>+</sup>18], which introduced novel algorithmic and arithmetic manipulation of the kernels for efficient computation. STOMP uses an ordered search scheme and introduces a streaming-dot-product [Nes23] that benefits from the locality of searches in matrix profile computation, which, in the end, results in cheap similarity distance computation among time series windows. This reduces the computational costs of STOMP to  $O(n^2)$  (see the discussions in Chapter 2). Despite this improvement in computational costs compared to STAMP, STOMP sacrifices the so-called anytime property in STAMP, where an algorithm with the anytime property is able to provide a close approximation to the exact matrix profile during computation (STAMP has this property). This property can be very useful in case an interactive user experience is required. SCRIMP++ resolves this issue. It first creates an approximation of the matrix profile using an algorithm (PreSCRIMP) that exploits Consecutive Neighborhood Preserving of time series to get a fast initial approximation. It then iteratively refines the approximation towards the exact solution, using the same tricks in STOMP. Overall, SCRIMP++ has a similar computational cost to STOMP.

The above algorithms solidified the foundation of methods for computing the matrix profile with a cost of  $O(n^2)$ , which is a quadratic growth of computational costs with the length of time series  $n$ . Therefore, computing the matrix profile for large time series demands high computational power. To target this high demand, Zimmerman et al. extended the **STOMP** approach and developed **SCAMP** [ZKS<sup>+</sup>19], a cloud-based framework for the parallel calculation of matrix profiles on multiple GPUs. **SCAMP** is known as the most efficient method to compute (single-dimensional) matrix profiles. On the same timeline, and as part of the development of this work, Pfeilschifter [Pfe19] developed **MP-MPI**, which is the first approach targeting matrix profile computation on HPC systems exploiting MPI model. We will elaborate further on this work in the next chapters.

#### 3.3.1 Streaming (Online) and Approximate Methods

All the above-mentioned algorithms target one-dimensional time series and batch computation. Also, except for **PreSCRIMP** (and **SCRIMP++** before full convergence), all compute an exact matrix profile. **LAMP** [SZSF<sup>+</sup>19] uses a neural network model to approximate matrix profile. Although this approach achieves significant speedups in comparison to exact solutions, it targets computation of matrix profile for data streams in *real-time* settings and, therefore, is not ideal for batch processing of large-scale datasets. Moreover, the authors clarify that the accuracy (e.g., false-positive rates) significantly depends on the quality of a reference dataset used to train the model, which is not ideal for real-world use cases. Recently, Lu et al. [LWM<sup>+</sup>22] developed the **DAMP** method, which targets anomaly detection use cases in time series streams. **DAMP** exploits the special case of anomaly detection to estimate bounds for matrix profile computation and prunes the computation, resulting in an approximate matrix profile that still accurately encodes anomalies.

#### 3.3.2 Reduced- and Mixed-Precision Approximate Methods

Also, there have been a few research items about the approximate computation of the matrix profile using reduced-precision computation: Zimmerman et al. [ZKS<sup>+</sup>19] investigate single-precision computation for single-dimensional time series, and Fernandez [Fer19] investigate matrix profile computation using FlexFloat [TMB20].

#### 3.3.3 Multi-Dimensional Methods

In 2017, Yeh et al. [YKK17] extended the matrix profile concept to multi-dimensional time series and developed the **mSTAMP** algorithm. **mSTAMP** is built on top of previous algorithms and is an iterative method involving streaming dot products (inherited from **STOMP**) to reduce computational costs. **mSTAMP** is also included in the powerful and scalable Python library, **STUMPY** [Law19].

### 3.3.4 Matrix Profile on Accelerators

**STOMP** and **SCAMP** are designed to utilize **GPUs** as their primary platform targeting the computation of the matrix profile for single-dimensional time series. Recently, Romero et al. [RVR<sup>+</sup>20] introduced a new approach called **ScrimpCo** for matrix profile computation on **FPGA**-based heterogeneous systems. This approach also targets single-dimensional time series. Recently, Kalantar et al. introduced **FA-LAMP** [KZB22], which aims at deploying the **LAMP** method on **Alveo FPGAs** for cloud settings as well as low-cost Xilinx **Ultra96** at Edge.

### 3.3.5 Key Implications of Reviewing the Work in Literature

Most of the methods for computing the matrix profile in the literature are based on the solidified formulations presented in **STOMP**, and **SCRIMP++** algorithms leading to  $O(n^2)$  computational costs. Also, there has been little attention to computational methods targeting the matrix profile for multi-dimensional time series, and no comprehensive studies on various computation aspects in multi-dimensional cases exist either. Finally, only a few works address the aspects of computation with approximation and reduced- and **mixed-precision**. The contributions of this thesis address these topics.

## 3.4 Summary

In this chapter, we covered an overview of work in the literature. We took a step back and looked at existing work on sable methods for time series mining. We then looked at the history of the evolution of the matrix profile concept and methods to compute it. We also covered the various aspects of methods to compute the matrix profile, including approximation, reduced- and **mixed-precision**, and multi-dimensionality. In the end, we discussed the key implications achieved by reviewing the work in the literature and highlighting the research gaps.

**Part II**

**Methods and Evaluations**





## 4 Time Series Similarity Mining at Extreme Scale on HPC Systems

The approach described in this chapter was previously published in ISC High Performance'20 [RKY+20], where the author of this thesis made major contributions to the contents, including supervising the development of a master thesis, the development of prototype codes, and evaluations. This chapter is a revised version of the aforementioned article reproduced with minor corrections and improvements.

### 4.1 Motivation

One important aspect in understanding multi-dimensional time series, and therefore getting insights into the physical systems behind the data, is the explorative discovery of similar and repeating patterns in a (potentially large) dataset [CEF+]. Recent advances in data mining approaches and techniques, specifically the matrix profile approach, enable the extraction of complex pattern structures and similarities in multi-dimensional time series. Yeh et al. [YKK17] extended the matrix profile approach for motif discovery in multi-dimensional settings. They also presented the first algorithm (**mSTAMP**) to compute the matrix profile for multi-dimensional time series, offering detailed insights into repeating patterns within and across dimensions of multi-dimensional time series.

Similar to the single-dimensional case, the **mSTAMP** approach is also compute-intensive, but the dimensionality adds another costly factor. Computational costs for multi-dimensional case scales with the number of dimensions. Therefore, for time series with a large number of records and high dimensionality (which is common in realistic scenarios), the matrix profile computation is no longer feasible on commodity systems. Additionally, the multi-dimensional matrix profile approach has not been shown to scale to larger systems, nor can it be used on anything but small datasets with low dimensionality.

However, multi-dimensional time series with large numbers of records in each time series are typical in many disciplines. One example is the operation of *industrial gas turbines* (Use Case 1 described in Section 2.1). Such systems are monitored by more than 100 different sensors and generate millions of records per month [KWFH+19]. Another real-world example is the monitoring of **HPC** infrastructures (Use Case 2 described in Section 2.1). Netti et al. [NMA+19] use up to 3176 sensors per node to monitor multiple production **HPC** systems at the *Leibniz Supercomputing Centre* and have shown that the collected monitoring data contains valuable information on the system's behavior, and can be used, e.g., in the characterization of applications running on the supercomputers.

To apply the concept of the matrix profile to large-scale multi-dimensional time series and hence to such real-world problems, we require new approaches to scale the computation of matrix profiles both to larger computational resources and to larger datasets.

In this chapter, we first take one step back and introduce a novel approach to compute the matrix profile in parallel on HPC systems, targeting large time series, for both single and multi-dimensional cases.

This approach is motivated by the basic observation that the calculation of a matrix profile is highly memory bound [Pfe19], and therefore, it can benefit from horizontal scaling of memory bandwidth. This approach addresses the complete computation of the matrix profile, including I/O, initialization, workload distribution, computation, and aggregation of final results. We demonstrate that parallel computation of the matrix profile can exploit the capability of high-performance interconnects and remain highly scalable. However, in order to achieve efficiency, a series of algorithmic advances and optimization steps are needed, which we introduce in this work and verify with an analytical performance model.

### 4.2 Research Questions

In particular, we are interested in addressing the following research questions (RQs):

- **RQ:** how can we characterize the matrix profile computation for multi-dimensional time series? Is it an I/O-, network-, memory-, or compute-bound computation?
- **RQ:** how can we design and apply efficient partitioning schemes for parallel computation of multi-dimensional time series?
- **RQ:** how far does the matrix profile computation scale on HPC systems? How can we characterize the benefits of HPC systems and hardware for matrix profile computation?

To answer these questions, in this chapter, we ...

1. ... introduce a new highly-parallel approach called,  $(MP)^N$  to compute the multi-dimensional matrix profile on HPC systems based on MPI. We use  $(MP)^N$  as the vehicle to answer the research questions posed above.
2. ... provide an analytical analysis for the computational costs and performance of this approach on HPC systems.
3. ... use experimental evaluations to demonstrate the scalability and efficiency of matrix profile computation on the SuperMUC-NG system.

### 4.3 Multi-Dimensional Parallel Matrix Profile: $(MP)^N$

We introduce a new approach for computing the multi-dimensional matrix profile in parallel. Building on top of the existing mSTAMP and STOMP approaches (see Chapter 2

### 4.3 Multi-Dimensional Parallel Matrix Profile: $(MP)^N$

and Chapter 3), we introduce  $(MP)^N$ , which stands for Multi-dimensional Parallel Matrix Profile. Similar to *STOMP* (described in Section 2.3.5),  $(MP)^N$  computes the distance matrix (in fact, the Pearson’s correlation factors) in an iterative in-place computation scheme (using the streaming dot product shown in Equation 4.5).

$(MP)^N$  adopts *mSTAMP* formulations for parallel processing, allowing the distribution of computational workload among multiple workers with minimum communication during the computation phase.  $(MP)^N$  partitions the time series along the records and distributes the workload across multiple processing elements, e.g., cores of a multiprocessor.  $(MP)^N$  leverages *MPI* for communication and parallel *I/O* to utilize the high-performance interconnect and *I/O* capabilities of *HPC* systems. With that, it exploits the computational power of *HPC* systems to compute the matrix profile for multi-dimensional datasets efficiently.

For a time series with  $d$  dimensions, each of length (number of records)  $n$ , and considering a subsequence length  $m$ ,  $(MP)^N$  computes a total of  $d$  distance matrices, one for each dimension. The distance matrix can be considered 3D, and it embeds the Euclidean distances for all subsequence pairs of the input time series in separate planes of a 3D matrix, one plane for each dimension. In the case of *self-join* settings, the distance matrix is symmetric.  $(MP)^N$  explicitly takes the symmetric structure of the distance matrices into account to avoid redundant computations. Further,  $(MP)^N$  complies with the semantics of the multi-dimensional matrix profile presented by Yeh et al. [YKK17]: In the multi-dimensional setting, sub-dimensional subsequences can correlate with any other subsequence in any  $k$ -dimensional ( $k \leq d$ ) subspace across all dimensions (see Section 2.3.3). Therefore, similar to *mSTAMP*,  $(MP)^N$  finds the minimum value of each column in the 3D distance matrices after sorting and partially aggregating the resulting distance values of the  $d$  dimensions.  $(MP)^N$  sorts the distances across the dimensions for all records in the distance matrix using an optimized memory layout combined with a high-performance sorting kernel. Note that similar to other *STOMP*-based approaches,  $(MP)^N$  avoids storing the full 3D matrix.

The resulting matrix profile represents the nearest neighbors of a subsequence with the best matching  $k$  ( $k \leq d$ ) dimensions (see Section 2.3.3).

#### 4.3.1 Mathematical Formulations of $(MP)^N$ Approach

We adapt the formulations of the *STOMP* approach presented in Table 2.3 to the multi-dimensional case and combine it with the *mSTAMP* approach. Table 4.1 presents this. We avoid repeating the description of the details of individual formulas in Table 4.1 since we already covered most of them in Section 2.3.5 for the single-dimensional case. Here, we mainly discuss the extensions to the multi-dimensional case. One difference between Table 4.1 and Table 2.3 is that here we have a 3D indexing, which corresponds to the multi-dimensional case (the third dimension is to index the different dimensions). More importantly, here we have additional formulas (Equation 4.8 and Equation 4.9) to sort the distance matrix along the dimensions.

In the multi-dimensional case, the streaming dot product works as follows, starting with an initialization step: for a multi-dimensional input time series, for each dimension,

---

Average of samples in a subsequence starting at record $i$ in dimension $k$	$\mu_{i+1,k} = \mu_{i,k} + (T_{i+m,k} - T_{i,k})/m$	(4.1)
Inverse of norm-1 of samples in a subsequence starting at record $i$ in dimension $k$	$d_{i,k}^{-1} = \left( \sum_{z=0}^m T_{i+z,k} - \mu_{i,k} \right)^{-1}$	(4.2)
Intermediate values used in mean-centered streaming dot product formulation 4.5	$df_{i+1,k} = (T_{i+m,k} - T_{i,k}) / 2$	(4.3)
Intermediate values used in mean-centered streaming dot product formulation 4.5	$dg_{i+1,k} = (T_{i+m,k} - \mu_{i+1,k}) + (T_{i,k} - \mu_{i,k})$	(4.4)
Streaming dot product of samples in subsequences $i$ and $j$ in $k$ -th dimension [ZKS <sup>+</sup> 19]	$\overline{QT}_{i+1,j+1,k} = \overline{QT}_{i,j,k} + df_{i,k} \times dg_{j,k} + df_{j,k} \times dg_{i,k}$	(4.5)
Pearson correlation matrix among subsequences starting at records $i$ and $j$ in dimension $k$	$\rho_{i,j,k} = \overline{QT}_{i,j,k} \times d_{i,k}^{-1} \times d_{j,k}^{-1}$	(4.6)
Euclidean distance matrix among subsequences starting at records $i$ and $j$ in dimension $k$	$D_{i,j,k} = \sqrt{2 \times m \times (1 - \rho_{i,j,k})}$	(4.7)
$i$ -th distance profile in dimension $k$	$\delta_{i,k} = D_{i,*,k}$	(4.8)
Sort $i$ -th distance profile along dimensions	$\delta'_i = \text{sort}(\delta_i)$	(4.9)
Average based on prefix sum of $i$ -th distance profile along dimensions	$\delta''_i = \text{prefix\_avg}(\delta'_i)$	(4.10)
Matrix profile and its index	$P_{i,k} = \min(\delta''_{i,k}), \quad I_{i,k} = \text{argmin}(\delta''_{i,k})$	(4.11)

---

**Table 4.1:** Iterative STOMP formulation extended for multi-dimensional matrix profiles.

the vectors  $\mu$ ,  $d^{-1}$ ,  $df$ , and  $dg$  (Equations 4.1- 4.4) are computed. This computation scales linearly with the number of records (i.e.,  $O(n \times d)$ ) and, therefore, is a cheap part of the computation. These vectors have the same semantics as in the single-dimensional case (discussed in Section 2.3.5); however, here, they are computed for each individual dimension.

After the initialization step based on Equations 4.1- 4.4, the (MP)<sup>N</sup> workload consists of the following iterative steps:

1. computing the distance matrix in multiple dimensions using the streaming dot product formulation (Equations 4.5- 4.7),

### 4.3 Multi-Dimensional Parallel Matrix Profile: (MP)<sup>N</sup>

---

**Pseudocode 3** (MP)<sup>N</sup> ( $\mathbf{T}$ ,  $\mathbf{m}$ ): core procedure for computing the multi-dimensional matrix profile.

---

**Input:** d-dim. time series  $T \in \mathbb{R}^{d \times n}$  and subsequence length  $m \in \mathbb{N}$ , and exclusion parameter  $e = m$ .

**Output:** multi-dimensional matrix profile  $P \in \mathbb{R}^{d \times n - m + 1}$  and matrix profile index  $I \in \mathbb{Z}^{d \times n - m + 1}$ .

---

```

1:  $\mu, d^{-1}, df, dg \leftarrow \text{precompute\_statistics}(T, m)$  ▷Equations 4.1– 4.4
2:  $QT \leftarrow \text{initialize\_streaming\_dot\_product}(T, m)$  ▷see [ZKS+19]
3: for  $i \leftarrow 0$  to  $n - m$  do
4:   for  $j \leftarrow i + e$  to  $n - m$  do
5:     for  $k \leftarrow 0$  to  $d - 1$  do
6:        $\delta_{j,k} \leftarrow \text{streaming\_dot\_product}(\mu, d^{-1}, df, dg, QT)$  ▷see Table 4.1
7:     end for
8:   end for
9:    $\text{handle\_exclusions}(\delta_{j,k})$  ▷see [YZU+16] and [YKK17]
10:  for  $j \leftarrow i + e$  to  $n - m$  do
11:     $\delta'_{j,*} \leftarrow \text{sort}(\delta_{j,*})$  ▷Equation 4.9
12:  end for
13:   $\delta'' \leftarrow \text{dimension\_wise\_prefix\_sum\_and\_normalize}(\delta')$  ▷see [YKK17]
14:  for  $k \leftarrow 0$  to  $d - 1$  do
15:     $P, I \leftarrow \text{element\_wise\_argmin}(P, \delta'')$  ▷matrix updates
16:     $\text{tmp} \leftarrow \text{row\_wise\_minimum}(\delta'')$ 
17:     $P_c[idx], I_c[idx] \leftarrow \text{element\_wise\_argmin}(P_c[idx], \text{tmp})$  ▷vector updates
18:  end for
19: end for
20:  $P, I \leftarrow \text{merge}(P, I, P_c, I_c)$ 

```

---

2. sorting and partially aggregating (using prefix sum<sup>1</sup> and averaging) the values in the distance matrices along the dimensions (Equation 4.9), and
3. updating the resulting matrix profile using element-wise *min* and *argmin* operations Equation 4.10.

In the case of a [self-join](#), we also exclude the trivial matches corresponding to the proximities of the diagonal entries in the distance matrices (similar to the single-dimensional case as discussed in Chapter 2).

Pseudocode 3 puts the formulas in Table 4.1 into a concrete procedure for computing the multi-dimensional matrix profile. This procedure can be summarized in the following steps:

---

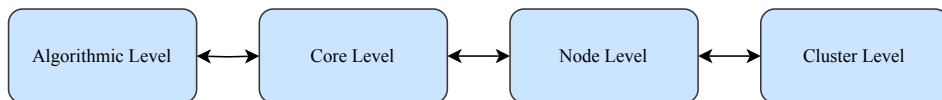
<sup>1</sup>Prefix sum, also known as the cumulative sum, and the inclusive scan, is the operation to compute the sums of prefixes (running totals) of the input sequence.

1. First, we (pre-)compute statistics of all subsequences using Equations 4.1—4.4 in all  $d$  dimensions (Line 1, Pseudocode 3).
2. In Line 2, Pseudocode 3, we initialize the streaming dot product with the distance between the *first* subsequence in the time series and all other subsequences. This is performed for each individual dimension.
3. We use Equations 4.5– 4.8 (Lines 4–8, Pseudocode 3) to iteratively—and in-situ—update the distance profile  $\delta$  (Line 3, Pseudocode 3) to avoid full storage of the distance matrices, by calculating the Euclidean distances between subsequence  $i$  and all the other subsequences  $j$ . This is done for all  $d$  dimensions. Further, in each iteration  $i$  of the main loop (Line 3), we . . .
  - a) . . .invalidate the entries of  $\delta$  corresponding to the exclusion zone (Line 9, Pseudocode 3),
  - b) . . .sort the distance profile  $\delta$  in each record and across the dimensions (Equation 4.9 and Lines 10–12, Pseudocode 3),
  - c) . . .prefix sum and normalize the values of the sorted distances in  $\delta'$  (Lines 13, Pseudocode 3), and
  - d) . . .find the distance profile using element-wise *min* and *argmin* operations, i.e., the matrix profile and its index are updated according to the values of  $\delta''$  computed in this iteration (Equation 4.10, Lines 14–18, Pseudocode 3).

#### 4.4 (MP)<sup>N</sup> for CPU-based HPC Systems

Targeting the (MP)<sup>N</sup> approach on modern CPU-based HPC systems requires addressing various design and optimization issues at various levels (see discussions in Chapter 2).

First of all, we ensure that we use algorithmic formulations with the lowest computational costs, high parallelization potential, and scalability. Next, at the core level, we prominently need to design data structures with suitable access patterns and data reuse and expose parallelism to higher levels. At the multi-core level, we need to design suitable, flexible, and tunable partitioning and tiling schemes to allow exploiting the parallelism on the node with multi-threading or multi-processing. Finally, at the cluster level, we prefer methods with minimal communication and synchronization operations during the execution of the main iteration loop of (MP)<sup>N</sup> (i.e., Line 3 in Pseudocode 3).



**Figure 4.1:** Design and parallelization of (MP)<sup>N</sup> at various levels.

#### 4.4.1 Algorithmic Design and Optimizations

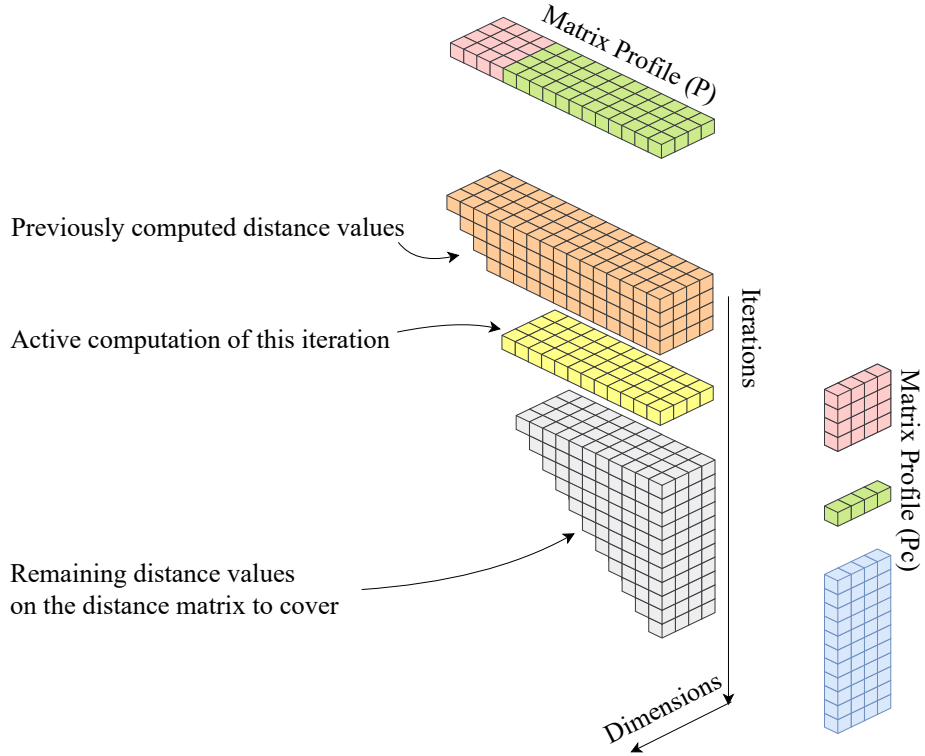
$(MP)^N$  follows the established techniques for computation of matrix profile (mainly applied to single-dimensional cases), extends and employs them for the multi-dimensional case. For the case of `self-join`, we employ a similar approach used in `SCAMP` [ZKS<sup>+</sup>19] and `MP-MPI` [Pfe19] to compute the upper triangular part of the distance matrices avoiding the redundant computation of the lower triangular part. In Figure 4.2, we illustrate these triangular distance matrices and their calculation (for a given iteration) by placing the corresponding matrices for various dimensions next to each other to shape a 3D upper triangular distance matrix (we remove the matrix profile index from the illustration for simplicity, but its structures and calculations resemble the ones depicted for matrix profile itself). This figure breaks down the distance matrix into three parts ...:

- The orange part on top illustrates the elements of the distance matrices that were already computed (and thrown away) in previous iterations,
- The yellow part in the middle illustrates the elements of the distance matrices that are being computed in the current iteration ( $\delta$  in Line 11 of Pseudocode 3),
- The white cubes in the lower part illustrate the remaining values to compute and cover in the remaining iterations.

The missing lower triangular parts can be compensated for by using the symmetrical structure of the distance matrices (only in `self-join`). However, since we use an in-situ computation and only store values corresponding to  $i$ -th iteration (i.e., the yellow elements in Figure 4.2 are stored, and the orange ones are dropped), we cannot explicitly transpose the distance matrix. Therefore, instead, an additional set of matrix profiles,  $P_c$ , and matrix profile indices,  $I_c$ , are introduced to capture partial computation of the matrix profile (see Line 17 in Pseudocode 3).  $P_c$  and  $I_c$  are computed by implicitly transposing the distance matrices employing the already computed distance values instead of computing the lower triangular parts.  $P_c$  and  $I_c$  are computed using `min` and `argmin` operations per dimension at each iteration on the distance profile  $\delta$ . The partial matrix profiles ( $P$  and  $P_c$ ) that are iteratively computed are also illustrated in Figure 4.2. In this illustration ...

- the elements colored with red are computed in previous iterations,
- green elements get updated in the current iteration,
- and the blue elements are yet to be computed in the following iterations.

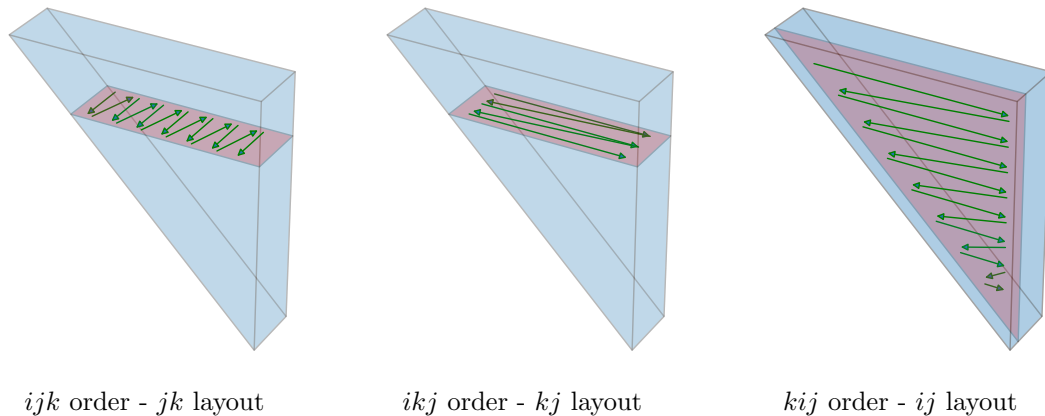
$P_c$  and  $I_c$  represent column-wise (for column  $c$ ) partial matrix profiles and indices, and similar to their partial row-wise counterparts, they are multi-dimensional. The final matrix profile and its index are constructed by merging the results of  $P_c$  to  $P$ , and  $I_c$  to  $I$  (Line 20, Pseudocode 3).



**Figure 4.2:** Illustration of 3D distance matrix (upper triangular part) and the iterative computation in  $(MP)^N$ .

In a naive approach, one can compute (and store) the full distance matrix using the nested loops in Lines 3-5 in Pseudocode 3. In such a naive computation, the order of the loops can be exchanged. In that case, all the 6 permutations of the loops ( $ijk$ ,  $ikj$ ,  $jik$ ,  $jki$ ,  $kij$ ,  $kji$ ) are valid (Figure 4.2 illustrates the  $ikj$  order). However, as we discussed before, the streaming dot product formulation, where the distance matrix is only partially stored (Lines 6 in Pseudocode 3), is often used in matrix profile computation to reduce computational costs as well as memory consumption. As we are using the streaming dot-product formulation, due to the data dependencies of this formulation, only a subset of loop permutations is feasible. Specifically, the  $ijk$  and  $ikj$  permutations are feasible (due to the symmetry of matrixes, we ignore  $jik$  and  $jki$  in discussions). The reason is that the permutations starting  $k$  loop require full storage of the distance matrix, which is both unwanted and infeasible—see the right-most sketch in Figure 4.3, where the full triangular matrix needs to be computed and stored for all the dimensions. However, a blocked version of these permutations ( $kij$  and  $kji$ ) are dropped from the analysis due to excessive memory footprint.





**Figure 4.3:** Selective combinations of the loop permutations and possible data layouts for storage in memory.

#### 4.4.2 Core-Level Design

An important issue to consider is the memory access pattern during the iterations. Note that the essence of the Pseudocode 3 is the iterative computation of  $\delta$  (again see Figure 4.2, where the yellow elements are computed in a certain iteration) followed by sorting and prefix sum, aggregation, and minimum computation. Therefore, the memory layout for storing  $\delta$  is the most important core-level design issue.

Having another look at Pseudocode 3, we notice that the loops in Lines 4 and 5 can be exchanged, and therefore, the layout for storing  $\delta$  is flexible from the point of view of the `streaming_dot_product` kernel. However, the loops at Line 10 and Line 14 have transposed-like access patterns.

We address the layout issue as follows: we select a layout that targets the most time-consuming kernels (e.g., `sort`, or `prefix_sum` or ...), i.e., we consider the more time-consuming kernels to be more impactful for the selection of the layout, and, therefore, we favor layouts that can help to improve these kernels. The other kernels, including the distance computation kernels (`streaming_dot_product`), are adapted to follow the selected layout. Our measurements show that regardless of the layout and choice of the sorting method, the most expensive part of the computation is the sorting kernel, and therefore, we always select a memory layout that provides stream access to the arrays and data used in the sorting kernel.

Figure 4.3 shows the possible layouts for storing  $\delta$ . The  $jk$  layout (Figure 4.3 left) is preferred in comparison to the  $kj$  (Figure 4.3 middle) layout, as it provides the sort kernels with stream-like access. As discussed before, the right-most one is not suitable as it requires an excessive memory footprint.

With the  $jk$  layout settled as the most efficient iteration layout, the most efficient iteration loop becomes the  $ijk$  order. We now look at the access patterns in various steps of the computation, the  $ijk$  order: for the distance computation, the  $ijk$  order results in stream-like access. Same for sorting, prefix sum, and normalization, where

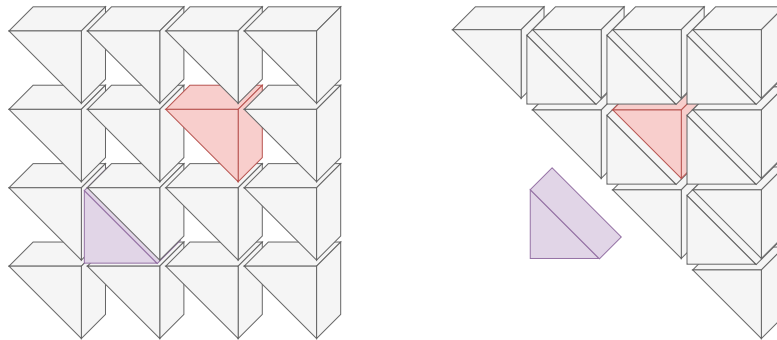
the  $ijk$  order has stream access. Finally, for the minimum computation, the  $ijk$  has a stride access. Therefore, for the *minimum* computation, we can use a double buffering scheme, where data is stored in a transposed fashion and allows for stream-like access. Additionally, the above-mentioned loop orders allow for jamming multiple  $j$  loops, which can improve data reuse.

#### 4.4.3 Node-Level Design and Parallelization

Mainstream HPC compute nodes have high-end multi-socket, multi-core processors and growingly more accelerators such as GPUs. In this section, we mainly focus on the  $(MP)^N$  design targeting multi-core processors.

To parallelize Pseudocode 3 on multiple cores, we need to address the  $i$  and  $j$  loop in Lines 3 and 4. If we take a close look at the structure of the iteration space in  $i$  and  $j$  loops (and ignoring the  $k$  loop for now), based on this, it resembles a 2-dimensional triangular iteration space. With this perspective, we can design a partitioning scheme for this iteration space by splitting it into smaller triangular partitions of iteration subspaces. Bringing the third dimension back to the picture, we end up with partitioning schemes with triangular prism-like tiles as depicted in Figure 4.4.

Figure 4.4 provides an illustration for two possible partitioning schemes that can be used in  $(MP)^N$  by decomposing the problem into 16 independent subproblems. In this illustration, we use a partitioning that splits the iteration space into 4x4 subproblems, resulting in 16 tiles, where the computation of a single tile (e.g., one tile is highlighted in red) can be done using the procedure  $(MP)^N(T, m)$  described in Pseudocode 4. The computation for all the omitted tiles can be eliminated due to the symmetry of the distance matrix (in the case of *self-join*). The computation for the tile in magenta is also omitted, similar to the rest of the missing tiles. However, we are highlighting it to illustrate the symmetry associated with the tile illustrated in red. Once the red tile is computed, the resulting partial matrix profile and index can also be used for the pink tile. In Figure 4.4, on the left, we illustrate the first partitioning scheme, where the distance matrix is split into smaller upper triangular prism tiles. On the right, in the second partitioning scheme, the whole lower part of the distance matrix is omitted. While both



**Figure 4.4:** Illustration for the partitioning scheme of the distance matrix in  $(MP)^N$ .

schemes are equivalent in the sense of computational costs, the one depicted on the left is slightly easier to implement as it does not require any change in Pseudocode 3. It only requires the implementation of kernels for computing upper triangular distance matrices, while the right scheme requires implementing and handling both upper and lower triangular kernels. Therefore, for simplicity, in (MP)<sup>N</sup> we rely on the left scheme. Regardless, in the case of cross-join, where the distance matrix is non-symmetric, we would need to implement kernels to deal with both the upper triangular and lower triangular tiles.

The aggregation step for the partial results from different tiles can be described as Equation 4.12 (Figure 4.4), where to aggregate the partial result  $P_{merged}^{(0)}$  from the first row of tiles, we combine the partial results associated with all the upper and lower triangular tiles. Here, we use  $(i, j)$  as a logical 2D index to identify the tiles. For example, the red tile in Figure 4.4 is located at index  $(1, 2)$ , and its symmetrical purple counterpart is located at  $(2, 1)$ .

$$P_{merged}^{(0)} = merge(P^{(0,0)}, P_c^{(0,0)}, P^{(0,1)}, P_c^{(0,1)}, P^{(0,2)}, P_c^{(0,2)}, P^{(0,3)}, P_c^{(0,3)}) \quad (4.12)$$

Due to the symmetry, the following property for all resulting matrix profiles in the tiles holds:  $P^{(i,j)} = P_c^{(j,i)}$  and  $I^{(i,j)} = I_c^{(j,i)}$ . Given this property, the partial matrix profiles for the problem illustrated in Figure 4.4 are merged according to Equation 4.13, which is equivalent to the Equation 4.12, only represented in terms of upper triangular tiles.

$$P_{merged}^{(0)} = merge(P^{(0,0)}, P^{(0,1)}, P^{(1,0)}, P^{(0,2)}, P^{(2,0)}, P^{(0,3)}, P^{(3,0)}) \quad (4.13)$$

These tiling schemes can be used independently of the node architecture, and in general, the tiles can be computed in parallel. Consequently, computing the matrix profile in parallel is similar to Pseudocode 3, except that it requires additional logic for tiling as well as extra logic to aggregate the partial matrix profiles computed in each tile as shown in Pseudocode 4.

Except for the aggregation part, the workload in tiles is embarrassingly parallel. Therefore, the computation of each tile can be assigned to a dedicated computing resource (e.g., a core). All the tiles can be computed in parallel using multi-threading or multi-processing. Note that the number of tiles ( $t^2$ ) is a squared number and can be set according to the available resources (e.g., cores) on the compute node.

In the case of multi-processing, the aggregation step requires handling of inter-process communication, which is handled similarly to the cluster-level design that is discussed in the upcoming section.

#### 4.4.4 Cluster-Level Design

As discussed before, the iterative computation of the distance matrix and matrix profile, based on the partitioning schemes introduced in Section 4.4.3 is an *embarrassingly-parallel* workload (except for the merging of the partial results). A similar partitioning

---

**Pseudocode 4**  $(MP)^N$ -tiling ( $T, m, t^2$ ): procedure to compute the matrix profile with tiling.

---

**Input:** The input time series  $T$ , the subsequence length  $m$ , and  $t^2$  number of tiles.

**Output:** The matrix profile  $P$  and its indexes  $I$ .

---

```

1:  $tile\_list \leftarrow compute\_tile\_list(T, t^2)$ 
2: for each  $tile \in tile\_list$  do in parallel
3:    $P^{tile}, I^{tile} \leftarrow (MP)^N(tile, m)$ 
4: end for
5: for each  $tile \in tile\_list$  do
6:    $P, I \leftarrow merge(P^{tile}, I^{tile})$ 
7: end for

```

---

and aggregation scheme can be employed in the case of distributed memory and multi-node computation: we use the same partitioning scheme and assign tiles to processes on the same or different compute nodes, i.e., each rectangular prism tile in Figure 4.4 will be assigned to a separate process. For the illustration in Figure 4.4, 16 processes running on 16 cores (or nodes) can run the computation in parallel.

In essence, each process has to compute partial matrix profiles by computing the first iteration using a naive sliding dot product formulation [YZU<sup>+</sup>16] (Figure 4.2), followed by sorting and prefix summation of distance matrix (again see Pseudocode 3 for details). Note that  $(MP)^N$  computes the matrix profile by evaluating only the upper triangular prisms of the partitioned distance matrix (*self-join*).

The first step (Line 1 in Pseudocode 4) consists of generating a logical grouping of processes for communication as well as creating the tiles and initializing their corresponding local data for computation. This tile creation step, together with aggregation step (Line 6 in Pseudocode 4), both require intra- or inter-node communication, which in the former case is mostly realized using broadcast operations and in the latter case is realized using reduction operations.

## 4.5 Putting it all Together – $(MP)^N$ Implementation in MPI

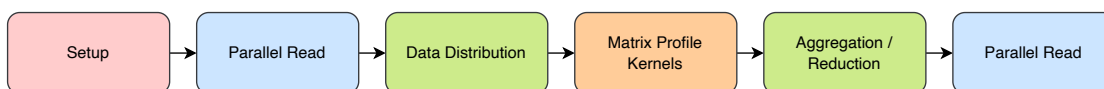
To fully utilize the resources of HPC systems, we implement  $(MP)^N$  using an end-to-end parallelization of all the steps from the setup phase to the final results steps (see Figure 4.5). We implement the parallel version of  $(MP)^N$  algorithm using the Message Passing Interface (MPI), which allows us to exploit the high-speed interconnect in a tightly coupled HPC systems<sup>2</sup>.

In particular, we avoid filesystem-based final aggregation [ZKS<sup>+</sup>19] and instead use MPI reduction operations, exploiting the high-speed interconnect. Furthermore, we

---

<sup>2</sup>See Appendix A4 for pointers to the code.

#### 4.5 Putting it all Together – $(MP)^N$ Implementation in MPI



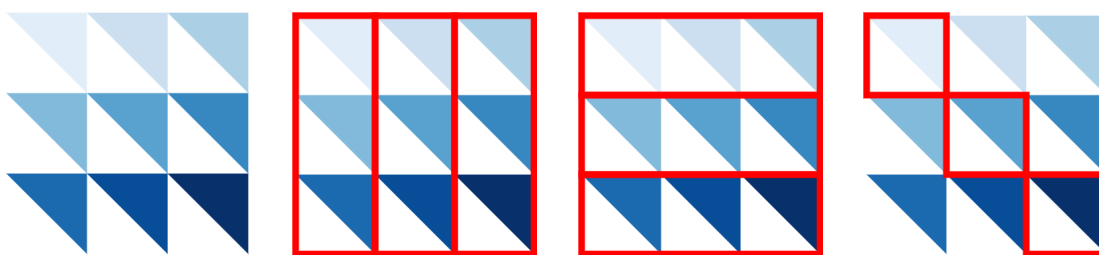
**Figure 4.5:** Illustration of all phases in our approach on a distributed memory HPC system.

utilize parallel MPI I/O functionality for massively parallel pre- and post-processing, and enable unprecedented scaling, all of which can be bottlenecks in HPC systems.

##### 4.5.1 Phases of $(MP)^N$

$(MP)^N$  implementation includes six phases (see Figure 4.5):

1. *Setup Phase:* We first build a virtual topology of processes (see Figure 4.6), in a 2D grid. This topology is used to create the MPI communicators required for reading and writing input and output time series from the file system, as well as the MPI communicators required for the distribution of time series and the aggregation of the resulting matrix profile. Finally, we distribute the data and workload among all MPI processes according to the partitioning schemes introduced in Section 4.4.3.
2. *File Read Phase:* we structure the storage of time series data in a custom “row-wise” layout (i.e., the data points from different data streams with a given timestamp are stored consecutively in the file), which is later on accessed by the processes in parallel. This strategy allows for simple partitioning with minimal parallel accesses to the file system (i.e., preventing bandwidth saturation as well as metadata bottlenecks), where only a subset of the processes (or nodes) perform I/O.
3. *Data Distribution Phase:* Processes that participate in the *File Read Phase* broadcast the input time series via the MPI communicators created during the setup.
4. *Kernel Execution Phase:* we execute the necessary kernels (i.e.,  $(MP)^N_{\text{tilled}}$ ) to compute a local multi-dimensional matrix profile for partitions of the input series.



**Figure 4.6:** Illustration of the virtual topology of MPI processes in communicators (left-most). Each triangle with a distinct color represents a separate MPI process working on a distinct part of the distance matrix. The red boxes represent exclusive process sets in communicators. The middle-left and middle-right figures represent communicators used in *Data Distribution* and *Aggregation* phases, and the right-most figure shows the communicator used in the I/O phases.

5. *Aggregation Phase*: we aggregate and merge the local results in each process to the final results (see Equation 4.12). This is done using reduction operations over MPI communicators created in the *Setup Phase*. After this phase, the final results are available on all processes that participate in the *Write Phase*.
6. *Write Phase*: finally, a subset of MPI processes (typically the processes located on the diagonals of the 2D grid topology) responsible for parallel-write operations outputs the final matrix profile and its index using MPI I/O (Figure 4.6).

### 4.5.2 MPI Communicators in (MP)<sup>N</sup>

We use three communicators in various phases presented in Figure 4.5. These communicators are illustrated in Figure 4.6 and are as follows:

- For reading and writing, we use only MPI processes responsible for the diagonals on the virtual topology (Figure 4.6 right-most). This allows us to reduce the pressure on the parallel file system caused by a large amount of I/O operations (*IOP*), which would otherwise degrade the I/O performance.
- We use both column- and row-wise communicators on the 2D virtual topology of processes for broadcasting data in the distribution step as well as the reduction of the partial results in the *aggregation* phase. We further use a custom *argmin*<sup>3</sup> MPI reduction operation<sup>4</sup>, according to the merging scheme described in Section 4.4.4, Equation 4.13 and Pseudocodes 3 and 4. This custom reduction is executed on the row-wise and column-wise communicators in the *Aggregation Phase*.

## 4.6 Limitations

The implementation of (MP)<sup>N</sup> has two main limitations:

1. The flexibility of the partitioning scheme: in our implementation, the tile sizes are coupled to the number of processes and the global problem size. This prevents exposing the tile size as a tuning scheme. However, the same partitioning scheme can still be adapted to work around this limitation to expose tile size as a tuning knob.
2. Our implementation only targets the *self-join* case and does not address cross-join between two distinct input time series. However, this is not a conceptual limitation, and the implementation can be extended to cover this case upon need.

None of the above-mentioned limitations are due to a conceptual limitation: these are rather limitations in the prototype implementation. In the next chapter, in the context of GPU acceleration, we will discuss a more flexible and general implementation. We

---

<sup>3</sup>See Chapter 2 for more details.

<sup>4</sup>MPI operation to reduce values on all processes to a single value.

will show that the same approach can be extended to support both flexible tiling and cross-join.

Despite these limitations, our evaluations demonstrate acceptable performance and high scalability of  $(MP)^N$ .

## 4.7 Evaluation

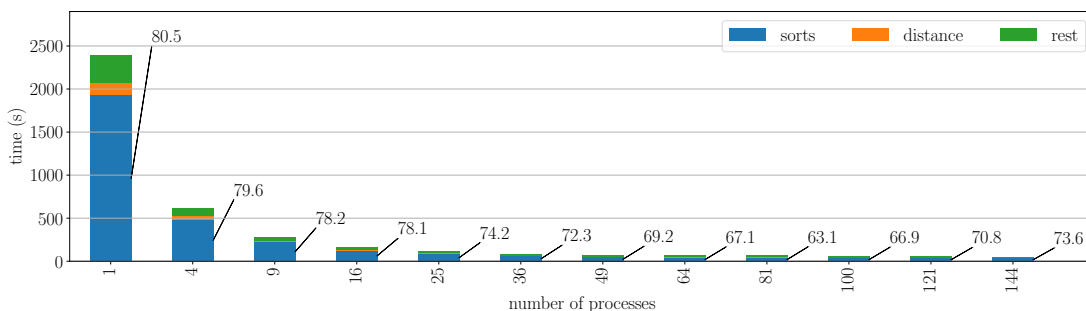
We setup a set of experiments to address the following questions:

- given the core-, node- and cluster-level designs, which subsystems bound the computation of the matrix profile? I/O, Network, memory, or compute?
- given the introduced partitioning schemes and the provided implementation, how much does the matrix profile computation scale?
- is using special HPC hardware beneficial for matrix profile computation?

### 4.7.1 Node-Level Performance Characterization

In order to characterize the performance of the matrix profile computation, we first look at single-node performance breakdown to identify the share of kernels from the run time. We then look at the performance breakdown for the phases once scaling beyond one node. The node-level performance breakdown helps us to discuss the main bottleneck kernels as well as whether the node-level code is bound by memory or cores. The larger experiments help to identify whether matrix profile computation is network- or I/O-bound.

**Single-Node Performance Breakdown** We insert instrumentation points to identify the time spent in various kernels. Specifically, we are interested in the most time-consuming kernels. We run strong scaling experiments on an Intel *Icelake* system in the *LRZ BEAST* testbed and show the breakdown of time spent in various kernels in



**Figure 4.7:** Performance breakdown of  $(MP)^N$  for a problem of size 64K and dimensionality of 128. The annotations show the percentage of time spent in sorting kernels. We report average values achieved via five repetitions for each experiment.

Figure 4.7. We consider a problem size with reasonable dimensionality (128) and number of records (64K).

We observe that the sorting kernel is the dominant kernel in all the configurations. While the distance kernel delivers the `flop/s` (and therefore, the `flop/s` rate is computed from that), still the performance of the code is dominated by sorts. Sort kernels are highly dominated by memory operations, and therefore, already at this point, we can conclude that ...

the matrix profile computation in multi-dimensional settings is memory-bound. Further investigation of `roofline` models also confirms this (not shown here. Also see work of [Pfe19]).

Note that the problem size certainly influences the plotted breakdown. For instance, pushing the dimensionality towards to lowest edge ( $d=1$ ) will eliminate the sorting portion. However, we observe this pattern (dominance of sort), even in small dimensionality (e.g., 16).

We also conduct another experiments to confirm this conclusion: we use `LK-WID` [THW10] to collect bandwidth utilization information and measured a maximum of  $\sim 140$ GB/s using 36 cores (18 per socket) on a single node of the SuperMUC-NG system<sup>5</sup>. This matches the achievable bandwidth reported by the `STREAM` benchmark [McC95] previously obtained on this machine<sup>6</sup>.

For the sorting kernel constitutes the prominent portion of run time, we focus on the sorting kernel and inspect alternative implementations of it. In particular, we compare three leading libraries for their performance:

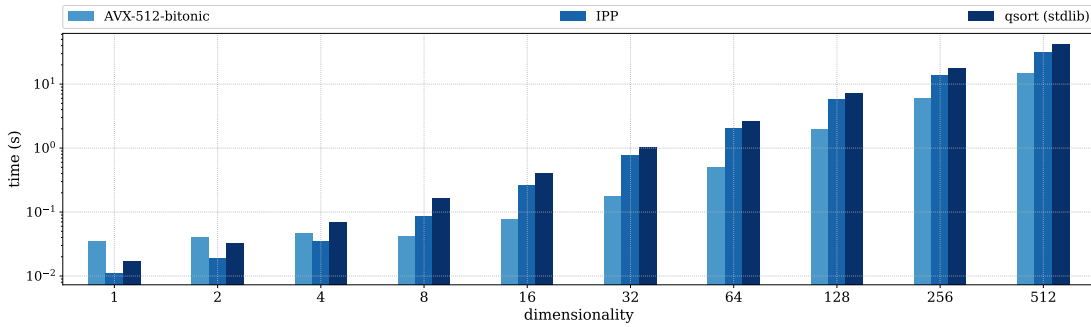
- `AVX-512-bitonic`, which is a high-performance sorting kernel based on the bitonic scheme [GKKKG03] in a library designed and implemented by Bramas [Bra17] targeting the sorting of small- and mid-sized arrays and is optimized for the Intel *Skylake* architecture,
- `Intel Integrated Performance Primitives (IPP)` [Tay], as a vendor-specific alternative library, and
- `qsort` from `C++-stdlib`-based [Jos12] as a basic Quicksort implementation.

We study the total run time of matrix profile computation, taking the sorting implementation and the dimensionality as parameters. We specifically look at the execution time on a single-core setup, which is sufficient (see Figure 4.7). In particular, we analyze the effect of the *dimensionality* parameter  $d$ . Figure 4.8 illustrates the result.

<sup>5</sup>(MP)<sup>N</sup> requires a quadratic number of MPI processes.

<sup>6</sup>We achieved a maximum bandwidth of 185.9 GB/s using `STREAM` benchmark for copy operation using 48 cores.





**Figure 4.8:** Comparing performance of various sorting kernels in  $(MP)^N$ . With  $n = 4K$  samples and  $m = 512$ . Each experiment is repeated 5 times, and we show the average result.

While all three kernels complete in similar time for smaller dimensionality, *AVX-512-bitonic* provides the best performance with increasing dimensionality.

The superiority of *AVX-512-bitonic* is in accordance with the results of an existing study by Bramas et al. [Bra17]. Also note that the y-axis on this graph is set to log scale, and therefore, the difference between the different implementations is very influential in practice. For this reason, we fix the *AVX-512-bitonic* kernel for all further experiments. Even the experiments discussed earlier in this Chapter use this sorting kernel.

Furthermore, we confirm an expected linearithmic<sup>7</sup> growth of execution time vs. increasing dimensionality parameter  $d$  for the three kernels, which validates the expected computational costs.

## 4.7.2 Characterizing Scalability

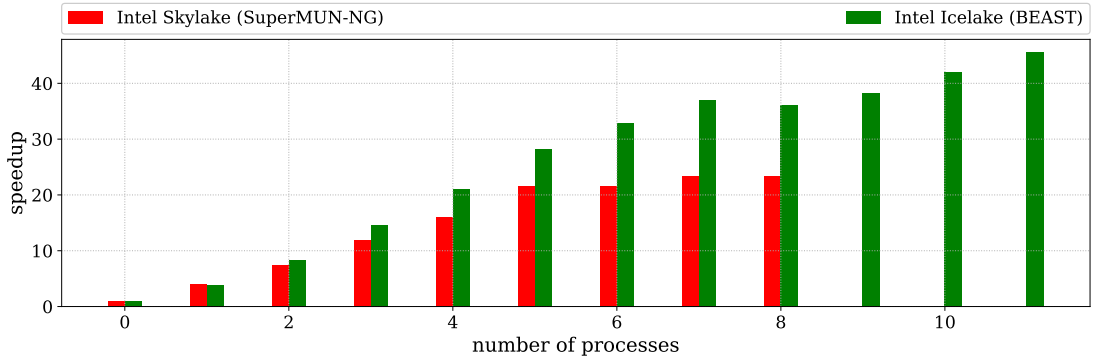
In order to characterize the scalability of the matrix profile computation, we first look at single-node scaling. Later, we use multi-node experiments to identify computation, network, and I/O loads.

**Single-Node Scalability** We present results from single-node execution on two systems. Both systems are from the Intel Xeon Processor Scalable Family. The first is an Intel *Skylake* (a single node of *SuperMUC-NG*), and the second one is an Intel *Icelake* (a single node in *LRZ BEAST*). The former experiment helps to characterize the scalability on *SuperMUC-NG* and the later helps us to validate the results on a more recent generation of Intel processors.

We execute  $(MP)^N$  with by enabling pinning the *MPI* processes within *NUMA* nodes<sup>8</sup>. As shown in Figure 4.9, we can see the expected performance saturation pattern for a memory-bound application in both systems. We further observe a performance increase

<sup>7</sup> $O(d \log d)$ .

<sup>8</sup>Specifically for Intel *MPI* we use: `I_MPI_PIN_ORDER=scatter`. For details on mapping *MPI* processes to cores, see <https://software.intel.com/en-us/mpi-developer-reference-linux-interoperability-with-openmp>.



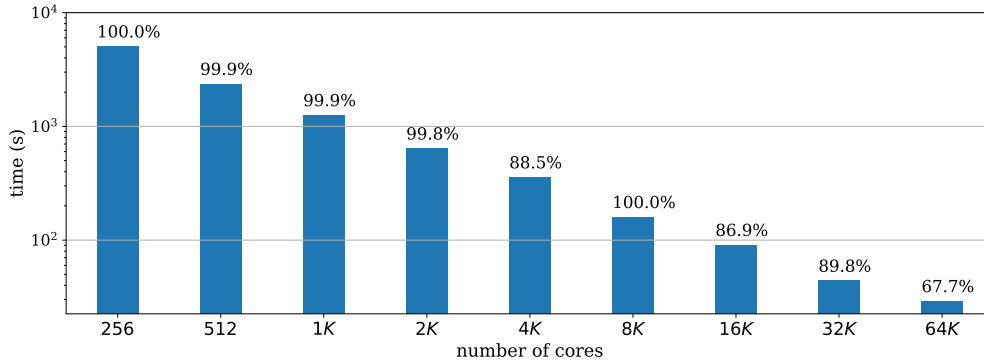
**Figure 4.9:** Saturation of performance in  $(MP)^N$  when increasing the number of MPI processes on a single Skylake node with 48 physical cores and  $(n = 10K, d = 128)$  vs. Icelake node with 72 physical cores  $(n = 64K, d = 128)$ . The structure of  $(MP)^N$  requires a squared number of processes. The bars with asterisks hatches run with hyper-threading.

when increasing the number of cores on a single node, but hyper-threading does not further improve the performance in both cases. The results in Figure 4.9 prove that the performance of  $(MP)^N$  is bound by memory bandwidth. Note that the iterations are highly dominated by sorting kernels that are full of memory operations. Also, as access to the underlying data is in a streaming fashion, there is little chance for optimizations, such as cache-blocking and utilizing smaller tiles, due to the lack of reusability and locality of the data.

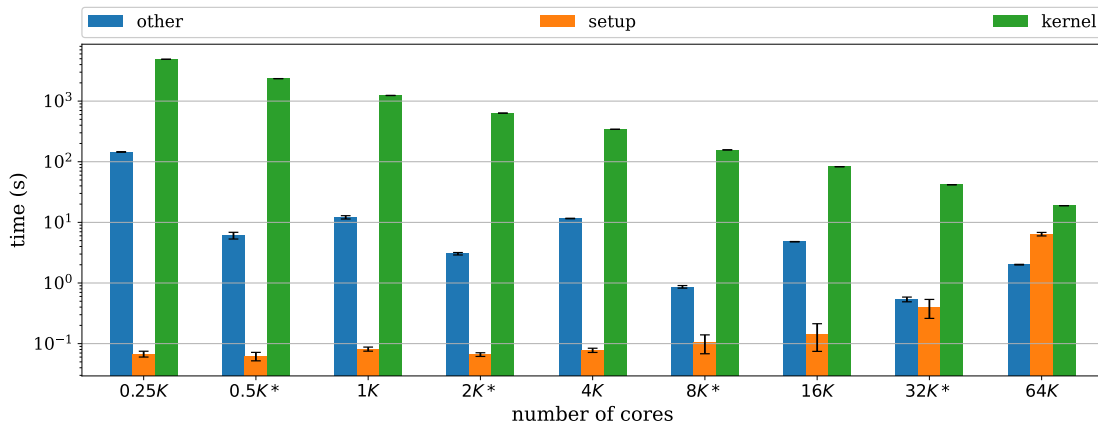
Single node performance of multi-dimensional matrix profile computations (using  $(MP)^N$ ) is bound by memory.

**Cluster-Level Scalability – Strong Scaling** We conduct strong scaling experiments on the SuperMUC-NG system using 6 to 1366 nodes (256 cores to 65536 cores). This corresponds to one-third of the full system. Figure 4.10 illustrates the speedup and efficiency for these experiments. We use time series of size  $n = 524288$ ,  $d = 128$ , with random contents and window size  $m = 512$ . We observe an acceptable scaling behavior with 67.7% efficiency on one-third of the system. While this scaling behavior remains acceptable, going to a larger number of cores will decrease the efficiency even more. Therefore, we present a breakdown of time spent in different phases of matrix profile computation.

Figure 4.11 presents this breakdown. We observe a linear speedup and throughput of the kernel execution time using the 6-node configuration as the baseline. However, the time for problem setup—mainly time for creating communicators—increases drastically. While this is very counter-intuitive, as the communicators are only constructed once at the beginning, still the communicator construction overhead grows drastically in com-



**Figure 4.10:** Results of strong scaling experiments for computing the matrix profile using  $(MP)^N$ . Parallel efficiency is annotated on top of the bars.

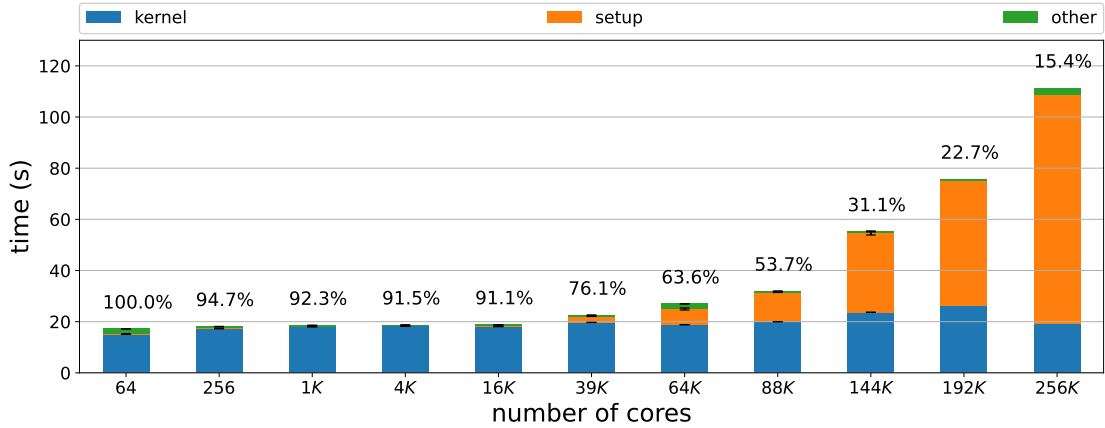


**Figure 4.11:** Detailed breakdown of time spent in various phases of matrix profile computation in strong scaling.

parison to kernel execution time in the strong scaling scenario. This reduces the parallel efficiency to 67.7% for the experiment with 1 366 nodes.

**Cluster-Level Scalability – Weak Scaling** Similarly, Figure 4.12 shows our results for weak-scaling experiments. Here, we fix the workload per core  $n_{core}$  to 2048. This results in a global problem size of  $n = 2048$  on 1 node with 1 core, scaling up to  $n = 1M$  on 5 462 nodes with 262 144 cores. The results of our weak scaling experiments show a kernel execution time of roughly 20 seconds. While in reality, we can have even larger problems and kernel execution time, this setup is enough to characterize the scaling behavior of  $(MP)^N$  and to illustrate dominating overheads. Again, we observe an ideal scaling of kernel execution time (remains roughly constant). However, we again encounter a significant increase in time spent on the creation of communicators, and therefore, the reduction of parallel efficiency to 63.7% on 64K cores (almost the same

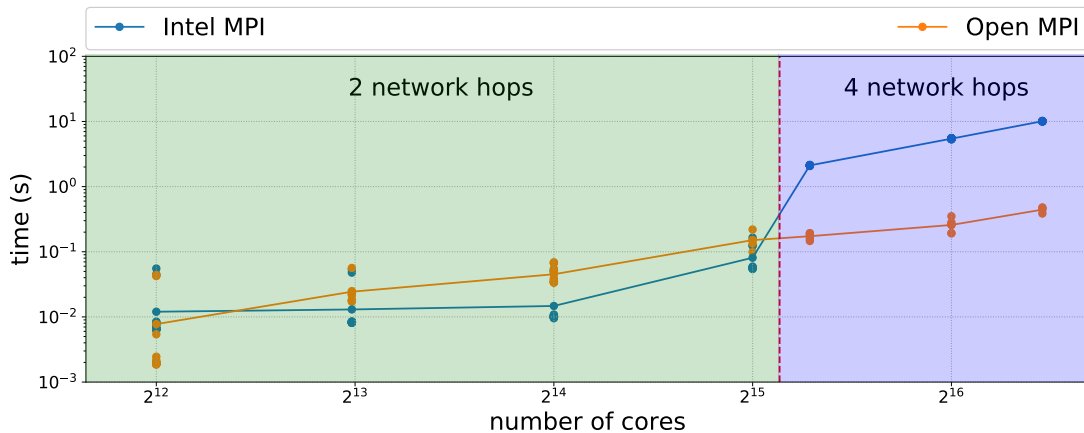
#### 4 Time Series Similarity Mining at Extreme Scale on HPC Systems



**Figure 4.12:** Detailed breakdown of time spent in various phases of  $(MP)^N$  for matrix profile computation in weak scaling. Parallel efficiency is annotated on top of the bars.

as strong scaling case) and down to 15.4% for the largest experiment, on almost 91% of the full system.

**Discussion on Scaling Overheads** While the presented scaling overheads seem very punishing, we can still justify them. As we observed, the main scaling overhead corresponds to setup. We argue that, since the setup time is independent of problem size and scales only with the number of MPI processes, for larger-scale problems, this overhead is less of an issue, as the presented overhead shows essentially the worst-case overhead. In reality, larger problems might be more of an interest, which results in larger kernel execution time compared to the setup phase and, therefore, renders better scaling efficiency. However, even though the run time can absorb the setup overheads in the more



**Figure 4.13:** Weak scaling of  $(MP)^N$ , illustrating the time spent in the kernels and the main time-consuming step, i.e., *Setup*.

common weak scaling scenario, it clearly represents a performance problem. Further, we conduct more investigations to understand the reason for this performance issue.

Another argument for the acceptability of the scaling behavior of  $(MP)^N$  is that we are using  $(MP)^N$  in **MPI**-only mode intentionally, as we aim at stressing the  $(MP)^N$  implementation to visit, investigate, and mitigate such issues. In **MPI**-only mode, utilizing 91% of the **SuperMUC-NG** translates to 262 144 processes over six islands in the fat tree topology (see Chapter 2 and Appendix A5.2 for details of the **SuperMUC-NG** network architecture) with a maximum of four hops distance between pairs. This setup, in fact, can raise some inefficiencies in an **MPI** implementation. The alternative would be to deploy the same algorithm using a hybrid mode (**MPI-OpenMP**). However, this might have other performance tuning and considerations, which we do not discuss in this work, as the current study can already justify the scalability of  $(MP)^N$ .

By inspecting the  $(MP)^N$  code, the measurements, and looking into the performance model presented by Raoofy et al. [RKY<sup>+</sup>20], we trace the inefficiency in the *Setup* phase to the implementation of vendor **MPI** functionalities rather than the implementation of  $(MP)^N$  itself or to hardware limitations. In particular, the overhead in this phase corresponds to the creation of communicators using `MPI_Comm_split`<sup>9</sup>. Subsequent experiments with an alternative **MPI** implementation (see Figure 4.13) have shown significantly better scaling for the `MPI_Comm_split` operation, indicating a non-scalable implementation by the initial **MPI**, which needs to be resolved. In addition to that, we can clearly observe the effect of the network topology of the **SuperMUC-NG** on the performance of the *Setup* step: we encounter a significant increase ( $\sim 58\%$ ) in time spent in the *Setup* step when we use four network hops of the fat tree interconnect (see the areas with light blue background color in Figure 4.13).

Overall, though, and despite the observed performance problem in **MPI**, the  $(MP)^N$  approach, we can make the following statement:

Cluster-level performance of multi-dimensional matrix profile computations (using  $(MP)^N$ ) is neither bound by file I/O nor network performance (assuming that the setup issue discussed above would be addressed) and is highly scalable. Therefore,  $(MP)^N$  can benefit from the horizontal scaling of compute resources in **HPC** systems.

Also,

The presented  $(MP)^N$  approach and its parallelization, including the partitioning scheme and its **MPI** implementation, can scale on a large portion of **SuperMUC-NG**.  $(MP)^N$  can validate the claims in the literature regarding the scalability of the matrix profile approach.

In particular, the  $(MP)^N$  approach is able to compute the matrix profile of a 128-dimensional time series dataset of one million records on the **SuperMUC-NG** petascale

---

<sup>9</sup>`MPI_Comm_split` partitions a group of **MPI** processes associated with a communicator into disjoint subgroups and creates a new sub-communicator for the subgroups.

system. This corresponds to a projected performance (with verified with measurements from Intel Advisor and LIKWID) of 1.3 petaflop/s for  $(MP)^N$  distance kernels [RKY+20].

## 4.8 Summary

In this chapter, we presented  $(MP)^N$ , the first scalable solution for the mining of large-scale multi-dimensional time series targeting CPU-based HPC systems. We discussed how existing single-dimensional approaches, as well as earlier efforts to compute the matrix profiles on large-scale HPC systems, are extended for this purpose.  $(MP)^N$  enables the computation of large matrix profiles—as a modern data mining approach—on an HPC system and makes it thereby applicable to large-scale real-world problems. We discussed that  $(MP)^N$  can leverage the abundant resources in HPC systems through horizontal scaling. We also demonstrated that this approach enables scaling up to 256K cores, providing highly scalable throughput. Further, with the experiments presented, we showed that the earlier claims in the literature regarding the scalability of the matrix profile approach do extend to the mining of multi-dimensional time series.

## 5 Time Series Similarity Mining on GPUs with Reduced- and Mixed-Precision

The approach described in this chapter was previously published in IPDPS'22 [JRY<sup>+</sup>22] and GTC'21 [JRS21], where the author of this thesis made major contributions to the contents, including supervising the development of a master thesis and guided research, and the development of the prototype codes, and evaluations.

### 5.1 Motivation

In the previous chapters, we established the importance of time series and, in particular, multi-dimensional time series that represent data captured by multiple sensor sources, as they include collective information about many components at the same time, as well as their mining in today's technological advances. We established how modern monitoring infrastructures provide scientists, data analysts, and developers with large amounts of data, which must be analyzed using big data and high performance data analytics techniques to better understand physical systems and phenomena. We also established the importance of the matrix profile method in mining multi-dimensional time series to enable cross-sensor correlations. Furthermore, we discussed the importance of introducing new efficient approaches such as (MP)<sup>N</sup> to unlock the potential of modern hardware as well as exploiting the power of HPC systems. In this chapter, we focus on extending (MP)<sup>N</sup> to another hardware powering up modern HPC systems, i.e., GPUs. This helps to push the technologies and capabilities in exploring the complex similarity patterns within multi-dimensional time series through matrix profile analysis, which is, in the end, crucial in extracting insights from the ever-growing deposit of datasets.

The previous studies in the literature, as well as our research on the computational aspects of the matrix profile (e.g., including the discussions presented in the previous chapter) show that for large time series datasets, the matrix profile requires a high computational power to evaluate the large distance (or equivalently correlation) matrices [ZZS<sup>+</sup>18, Pfe19, RKY<sup>+</sup>20]. The multi-dimensional case exposes even more computational costs: the computational costs scale with dimensionality, as each dimension requires the calculation of a separate distance matrix. Additionally, in the multi-dimensional case, we require extra repeated sort and prefix average (see Table 4.1) operations to connect the dimensions, which increases the computational cost further. All these aspects of computational costs for matrix profile computation make a good case for exploiting the parallelism and power of many cores in GPUs. In addition, we showed

in the previous chapter that matrix profile computation is memory-bound [Pfe19], suggesting that leveraging the High Bandwidth Memory (HBM) of GPUs promises performance improvements. Additionally, as this workload is neither communication-bound nor I/O-bound, the throughput is expected to scale with multiple GPUs.

However, the use of GPUs requires redesigning the parallelization scheme presented in the last Chapter: while a GPU-based multi-dimensional matrix profile benefits from the parallelization method used in the state-of-the-art GPU-based solution [ZKS<sup>+</sup>19] for the single-dimensional case, due to the extra computational costs, the resulting bottleneck shifts. Therefore, for the multi-dimensional case, significant changes in the parallelization scheme, as well as the redesign of data layout, distance, and sorting kernels, are required.

On top of that, modern GPUs are equipped with huge computational power when running/computing in reduced-precision arithmetic (see Chapter 2 for a brief introduction to such arithmetics). Therefore, the problem of finding similar-enough, but not necessarily exactly identical, patterns offers the door to reduced- and mixed-precision calculations, which have not been investigated before and can benefit from the capabilities of modern GPU hardware. Reduced- and mixed-precision computation schemes, aside from improving performance, can also reduce the memory footprint, resulting in an even more efficient usage of the GPU memory bandwidth and the ability to support larger problems. However, it does naturally lead to more numerical errors and, hence, new challenges to preserve acceptable numerical accuracy.

In order to address both computational costs and accuracy aspects of multi-dimensional matrix profile computation on GPUs, we, therefore, need ...

1. a careful design of a solution targeting GPU hardware, including an efficient data management design and careful design of compute kernels in various aspects such as the data layout,
2. an efficient sorting scheme suitable for GPUs,
3. a flexible tiling scheme allowing for various optimization and efficient multi-GPU parallelization and
4. a suitable arithmetic approach for efficient and sufficiently accurate reduced- and mixed-precision computation.

In this chapter, we extend the (MP)<sup>N</sup> approach and introduce a new extension for GPUs that addresses the mentioned data management, kernel, and arithmetic design challenges. This approach targets multiple GPUs and efficiently utilizes the GPU hardware features. We introduce reduced- and mixed-precision computation modes with improved arithmetics to support a sufficiently accurate computation in practice using single and half-precision arithmetics. We introduce an extension to (MP)<sup>N</sup> with a new tiling scheme that not only enables parallelization of the workload on multiple GPUs but also improves the accuracy of computation in reduced- and mixed-precision modes by bounding the numerical error propagation.



## 5.2 Research Questions

We are interested in addressing the following research questions (RQs):

- **RQ:** how much performance benefits do GPUs bring in compared to CPU-based implementation ( $(MP)^N$ ) for matrix profile computation?
- **RQ:** how to extend the partitioning scheme in  $(MP)^N$  to target nodes with multiple GPUs? Does the computation scale on multiple GPUs? Can matrix profile computation for multi-dimensional data be efficiently deployed on multi-GPU systems?
- **RQ:** how can we characterize the accuracy of matrix profile computation in reduced- and mixed-precision computation schemes for multi-dimensional time series in terms of mathematical and practical accuracies? How accurate are our solutions in terms of mathematical accuracy and also in practice?
- **RQ:** in practice, with reduced- and mixed-precision computation schemes, how much reduction in the accuracy can be traded for performance gain and vice versa? Does the tiling scheme play any role in the performance-accuracy trade-off?

To answer these questions, in this chapter, we ...

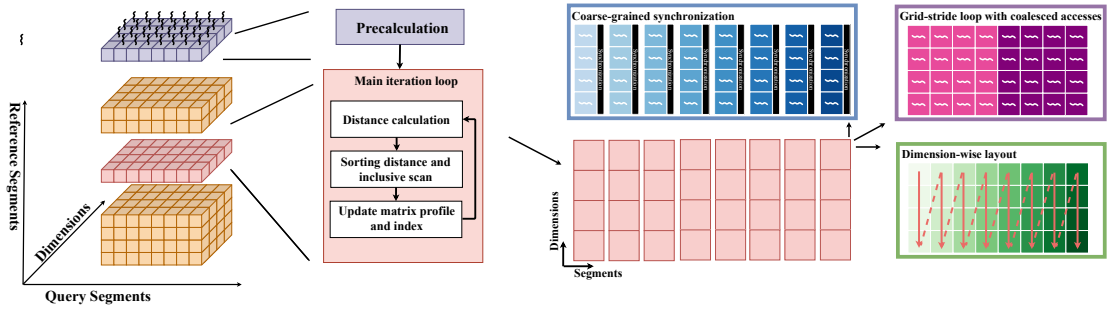
1. ... preset extensions of  $(MP)^N$  to compute the matrix profile for multi-dimensional time series on GPUs. For this, we extend the  $(MP)^N$  partitioning scheme into a more flexible tiling scheme to allow for further optimizations;
2. ... introduce several reduced- and mixed-precision modes for multi-dimensional matrix profile computation leveraging reduced-precision arithmetics on GPU hardware.

## 5.3 GPU-based Approach with Reduced- and Mixed-Precision

Our main targets in this chapter are high-end multi-GPU systems that are common in HPC centers and are growing in their attraction for High Performance Data Analytics (HPDA). The core concept behind our work is to accelerate the computation with the efficient exploitation of GPU hardware, multi-GPU parallelization, and reduced- and mixed-precision computation modes. We, therefore, extend the solution we presented in the previous chapter and refer to it as  $(MP)^N$ -GPU. Figure 5.1 illustrates the overview of this approach.

### 5.3.1 GPU-based Approach: $(MP)^N$ -GPU

We use the same equations listed in Table 4.1 and extend the Pseudocode 3. In fact, for GPU acceleration, the primary target is the efficient deployment of sorting kernels (we



**Figure 5.1:** An overview of single-tile matrix profile computation approach on GPU.

already established in the chapter that this is the dominant portion of the workload) and distance computation kernels (see Figure 5.1).

Using the formulations in Pseudocode 3 has several advantages, including cheap distance matrix computation on GPUs through using highly efficient arithmetic for the evaluation of distances between segments, i.e., streaming dot product formulation. Also, this formulation enables using the scarce GPU memory resources by taking advantage of partial computation and storage of distance matrix in GPU memory.

Figure 5.1 provides an overview of (MP)<sup>N</sup>-GPU, including the illustration of our kernels running on GPUs and the corresponding optimizations. We exploit an optimized data layout to enable coalesced memory accesses and optimized sort kernels. Additionally, we exploit the hardware features in modern GPUs to achieve high performance. Specifically, we tune the utilization of GPU Streaming Multiprocessors (SMs) and scratchpad memory<sup>1</sup> to achieve high bandwidth and low latency access in kernels.

### 5.3.2 GPU Approach (Single-Tile)

We start explaining our (MP)<sup>N</sup>-GPU, in Pseudocode 5, by providing a glimpse at a single tile GPU approach computation of a single GPU.

Pseudocode 5 describes the core algorithm of (MP)<sup>N</sup>-GPU that targets a single tile and single GPU. Note that (MP)<sup>N</sup>-GPU requires two input time series and works primarily for cross-join (but an extension to self-join is straightforward and is just an extra feature in the implementation). It starts with an asynchronous copy of the input time series data from the CPU (host) to the GPU (device), followed by the invocation of the compute kernels. `precalculation` kernel prepares the correlations  $QT$  associated with the first row of the distance matrix using a naive (non-streaming) dot product formulation. Additionally, this kernel computes the variables  $df$ ,  $dg$ ,  $\dots$ , used for the next  $n$  iterations using cumulative summation operations (corresponding to Lines 1 and 2 in Pseudocode 3). In more detail, each GPU thread computes one dot product (one element of  $QT$ ) and the corresponding cumulative summations for each element, illustrated in Figure 5.1 in purple.

<sup>1</sup>Scratchpad memory refers to a special high-speed memory used to hold small items of data for rapid retrieval. For NVIDIA GPU, it is often used to refer to the so-called shared memory.

### 5.3 GPU-based Approach with Reduced- and Mixed-Precision

---

**Pseudocode 5** (MP)<sup>N</sup>-tile ( $T_r^{cpu}, T_q^{cpu}, m$ ): Single-Tile GPU approach

---

**Input:** The reference and query time series  $T_r^{cpu}$  and  $T_q^{cpu}$ .

**Configuration:**  $s_{block}$  and  $s_{grid}$ .

**Output:** The matrix profile  $P^{cpu}$  and index  $I^{cpu}$ .

**Note:** All data resides on GPU unless marked otherwise.

---

```

1:  $T_r, T_q \leftarrow \text{input\_async\_cpy}(T_r^{cpu}, T_q^{cpu}, m, \text{H2D})$ 
2:  $QT_r, QT_q, df_r, dg_r, \dots \leftarrow \text{precalculation}(T_r, T_q, m)$ 
3: for  $i \leftarrow 0$  to  $(n - 1)$  do
4:    $\delta \leftarrow \text{streaming\_dot\_product}(\langle\langle s_{grid}, s_{block} \rangle\rangle)(QT_r, QT_q, df_r, dg_r, \dots)$ 
5:    $\delta'' \leftarrow \text{sort\_and\_prefix\_average}(\langle\langle s_{grid}, s_{block} \rangle\rangle)(\delta)$ 
6:    $P, I \leftarrow \text{update\_matrix\_profile}(\langle\langle s_{grid}, s_{block} \rangle\rangle)(\delta'')$ 
7: end for
8:  $P^{cpu}, I^{cpu} \leftarrow \text{output\_async\_cpy}(P, I, \text{D2H})$ 

```

---

The main iteration loop consists of the following steps: in the  $i^{\text{th}}$  iteration (Line 3 in Pseudocode 5), only one row (plane) of the distance matrices with size of  $\mathcal{O}(n \cdot d)$  is computed (the highlighted red plane in Figure 5.1). Note that out of the whole distance matrix, only the computation of the one row (plane) is active during an iteration, and only this part of the distance matrix is stored, and the updates are in-place. In this way, to process the time series with  $n$  subsequences and  $d$  dimensionality, only storing  $\mathcal{O}(n \cdot d)$  bytes of memory on GPU is required. In more detail, the iteration  $i$  include the following kernels:

- **streaming\_dot\_product** kernel uses the streaming dot product formulation (see Equation 4.5) to compute  $i^{\text{th}}$  rows of the distance matrix  $QT$  in-place. Each element of the new row (plane) of the distance matrix (i.e., the row that is computed in this iteration) is assigned to one GPU thread to compute it using Equation 4.5 in parallel. This effectively is the parallelization of Equation 4.5 in  $i$  (or  $j$ ) and  $k$ . This parallelization scheme and thread assignment are illustrated in Figure 5.1 in purple and pink.
- **sort\_and\_prefix\_average** kernel sorts the distances in ascending order along dimensions. For this kernel, multiple sorts (one for each dimension) are performed in parallel, where multiple threads cooperate on these sort operations: each individual sort operation is assigned to a group of threads. Additionally, this kernel performs the prefix averages (Equations 4.9 and 4.10 in Table 4.1) in parallel along the dimensions, where threads in each group cooperatively perform a prefix average. Due to the advantages of the parallel bitonic sort [GKKG03] that we discussed in the last chapter, we use this sort algorithm running in only  $\text{Log}^2(d)$  step delay. Also, for prefix average, we use a parallel fan-in approach in only  $\mathcal{O}(\log d)$  steps. The parallelization for this kernel, the thread and group assignment, and synchronization among threads for sorting are illustrated in Figure 5.1 in blue.

- `update_matrix_profile` kernel merges the computed distances in the  $i^{\text{th}}$  iteration to the results in previous iterations using Equation 4.11. This kernel uses a similar thread assignment as in the `precalculation` kernel, where all the threads update the elements of the resulting matrix profile (e.g., a plane) in an embarrassingly parallel fashion.

**Optimizations:** To achieve high efficiency on GPUs, we introduce the following optimizations in the above steps:

*Data Layout:* For storing the active 2D rows (planes) of the distance matrix in GPU memory, we use a lexicographical data layout with dimension-major (dimension-wise) order. Here, by dimension-major, we mean placing the consecutive elements of each dimension to reside next to each other in memory (shown in Figure 5.1 in green). Although this design primarily targets the sorting kernel, we employ this data layout for all the data involved in the computations of the different kernels. While this layout resembles the layout in the CPU implementation of  $(MP)^N$ , it can still be used to leverage coalesced memory access in `streaming_dot_product`, and `update_matrix_profile` kernels, and partly in `sort_and_prefix_average` kernels.

*Grid-Stride Loops:* We structure the iterations in kernels to exploit grid-stride loops to ensure coalesced memory access. Moreover, the grid-stride loops enable more flexibility by supporting arbitrary kernel launch configurations, i.e.,  $s_{\text{block}}$  and  $s_{\text{grid}}$ . Through this flexibility, our design promises a high performance through tuning kernel launch configurations (i.e., the configuration settings in Pseudocode 5) that match the GPU hardware architecture.

*Coarse-Grained Synchronization:* In `sort_and_prefix_average`, we use the  $\mathcal{O}(\log^2 d)$  parallel sorting scheme based on bitonic sort [GKKG03] and  $\mathcal{O}(\log d)$  parallel fan-in approach of prefix average, where many threads cooperatively sort and calculate the prefix averages. Compared to the more intuitive batch-based parallelization, where only one thread performs a single sort and prefix average, our choice results in better utilization of the GPU resources, hence achieving higher performance. However, the parallelization in this scheme requires nested synchronization (for bitonic sort), which can potentially lead to large overheads. To minimize the overhead, we apply coarse-grained synchronizations among subsets of threads in groups.

### 5.3.3 Multi-Tile GPU Approach

While the GPU-based approach in Pseudocode 5 presents an all-inclusive scheme, i.e., starting from time series stored on the CPU and ending with the resulting matrix profile in the CPU, still it has the following limitations:

- It is only targeting a single GPU, while in many HPC setups, nodes are equipped with multiple GPUs.
- Despite the efficient memory capacity requirements of streaming dot product formulation ( $\mathcal{O}(n \times d)$ ), the problem size (i.e., tile size) can be still limited by the

### 5.3 GPU-based Approach with Reduced- and Mixed-Precision

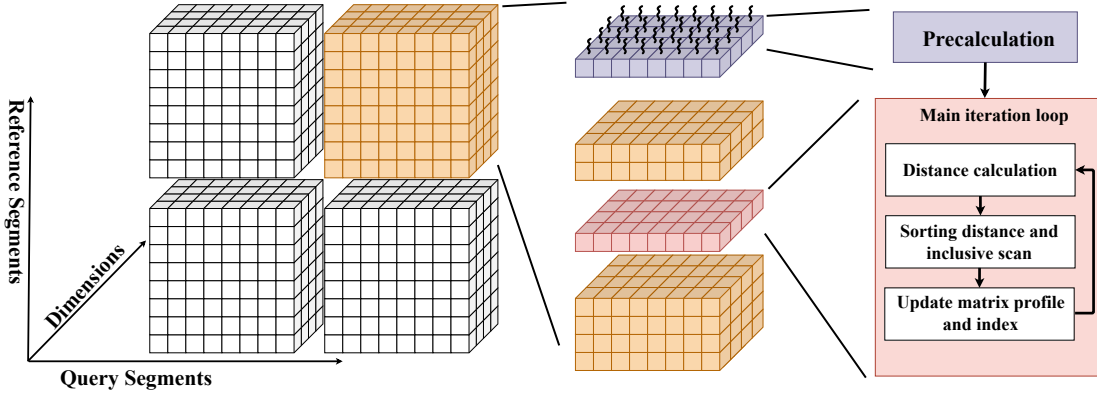


Figure 5.2: An illustration for the multi-tile matrix profile computation approach on GPUs.

amount of GPU memory capacity. This could impose severe limitations for cases with high dimensionality.

To address these limitations, and specifically, to exploit parallelization on multiple GPUs, and relax the memory requirements, we introduce a flexible tiling scheme  $((MP)^N\text{-GPU})$  that extends the partitioning scheme in  $(MP)^N$ . This tiling scheme (illustrated in Figure 5.2) is motivated by task-based programming models, where the computation of each tile is a stand-alone task. Therefore, tasks associated with these tiles can be computed in parallel: This scheme also *decouples* the size of the tile and, therefore, the size of running intermediate allocated buffers on the device, e.g., for storing distance matrices such as  $QT$  from the total size of the input series ( $T_r^{CPU}$  and  $T_q^{CPU}$ ) and the corresponding distance matrix  $\delta$ . Therefore, despite the limited device memory, this tiling scheme can process arbitrarily large problems (in the sense of both the number of segments and the number of dimensions) on a single or multiple device(s).

Figure 5.2 illustrates this tiling scheme. In this illustration, the total distance matrix is split into four tiles, each of which computes one tile with  $(MP)^N$  as presented in Pseudocode 5. These tiles can be executed sequentially by a single GPU, e.g., if there is only one GPU available on the node, or they can be assigned to multiple GPUs on the node to run in parallel. Also, we want to highlight the resemblance between the partitioning illustrated in Figure 5.2 and Figure 4.4. As we mentioned before, the main difference is that here, the tiles are cubic (as we address cross-joins here), and multiple tiles can potentially run on a single GPU, while there is a 1-1 relation between the number of cores and tiles in  $(MP)^N$  implementation for CPUs (Figure 4.4).

We describe the details of this tiling scheme in Pseudocode 6. We first split the distance matrix into smaller tiles (`compute_tile_list`), where each smaller tile is later executed on a GPU as a standalone matrix profile computation (task) with a smaller problem size (i.e., tile size). We statically assign these tiles (`assign_tile`) to  $n_{gpus}$  GPU(s) in a round-robin fashion, enabling a balanced load for parallel execution on multiple GPUs. The computation of tiles is triggered asynchronously, and they run on GPUs in parallel using the scheme presented in Pseudocode 5.1. After the execution of each tile, its results

**Pseudocode 6** (MP)<sup>N</sup>-GPU: Multi-Tile Approach

**Input:** The reference and query time series  $T_r^{tile}$ ,  $T_q^{tile}$  stored on CPU, and subsequence length  $m$ .

**Configuration:**  $s_{block}$ ,  $s_{grid}$ ,  $n_{tiles}$ , and  $n_{gpu}$ .

**Output:** The matrix profile  $P$  and its indexes  $I$ .

**Note:** All data resides on GPU unless marked otherwise.

---

```

1:  $tile\_list \leftarrow compute\_tile\_list(n_{gpu}, n_{tiles})$ 
2: for each  $tile \in tile\_list$  do in parallel with implicit synchronization
3:    $dev \leftarrow assign\_tile(tile)$ 
4:    $P^{tile}, I^{tile} \leftarrow (MP)^N\text{-tile}(T_r^{tile}, T_q^{tile}, m, s_{grid}, s_{block}, dev)$ 
5: end for
6: for each  $tile \in tile\_list$  do ordered
7:    $P^{CPU}, I^{CPU} \leftarrow merge(P^{tile}, I^{tile})$ 
8: end for

```

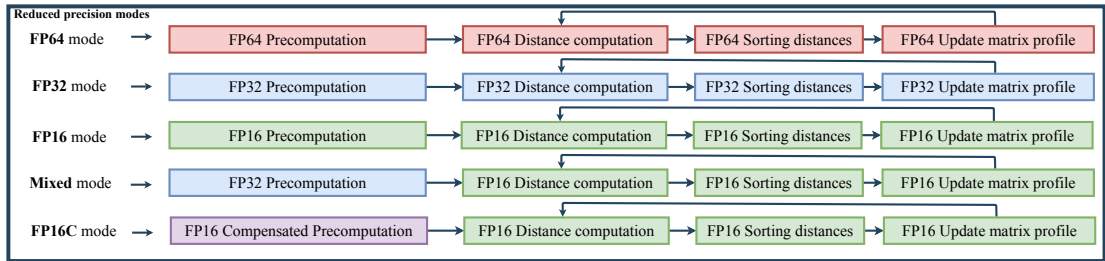
---

are merged (`merge`) on the CPU using the same merging scheme presented in Table 4.1. Data transfer and kernel execution for tiles benefit from implicit synchronization (i.e., CUDA Streams), to exploit maximal concurrency in kernel execution as well as to hide the communication latencies between CPU and GPU.

In addition to the kernel launch configuration itself, the number of GPUs ( $n_{tiles}$ ) and tiles ( $n_{tiles}$ ) are used as configuration parameters (i.e., the configuration settings in Pseudocode 6) for performance tuning.

### 5.3.4 Reduced- and Mixed-Precision Modes

Another approach to address the efficient computation of the matrix profile on GPUs is using reduced-precision arithmetics. Especially as modern GPUs provide special support for reduced-precision arithmetics and instructions. However, the streaming dot product formulation that is widely used in matrix profile computation methods (including (MP)<sup>N</sup> and (MP)<sup>N</sup>-GPU), due to its suitable costs, can introduce challenges for reduced-precision computation. This formulation uses prefix sums all over the place,



**Figure 5.3:** Overview of various precision modes for matrix profile computation provided in (MP)<sup>N</sup>-GPU.

### 5.3 GPU-based Approach with Reduced- and Mixed-Precision

which can suffer from numerical round-off errors. Therefore, a naive implementation for the streaming dot product and the prefix sum can lead to very inaccurate results when using reduced-precision computation. To investigate these challenges, we explain various precision modes covered in (MP)<sup>N</sup>-GPU from full double-precision computation down to half- and **mixed-precision** modes. Also, we investigate combining these modes with a simple tuning technique for configuring tile sizes: we leverage the tiling scheme in (MP)<sup>N</sup>-GPU to also limit the excessive propagation of numerical errors in reduced- and **mixed-precision** modes.

(MP)<sup>N</sup>-GPU natively relies on double-precision (**FP64**) floating point operands to ensure accuracy in multi-dimensional segment distances. To further improve the performance while ensuring accuracy on a certain level, we consider leveraging reduced-precision arithmetics in **precalculation** and **streaming\_dot\_product** kernels (see Figure 5.3). The reason for this choice is that these two kernels comprise the main computation costs that can be affected by reduced-precision computation. We introduce four additional reduced- and **mixed-precision** modes that use *single*-precision floating point arithmetic (**FP32**), *half*-precision floating point arithmetic (**FP16**), a *mix* of both, and *half*-precision mode with improved arithmetic in the **precalculation** step.

*Single precision (FP32)*: as **FP32** is widely used in **ML** and **HPDA**. For this computation mode, we use **FP32** in all steps for both storage and arithmetic.

*Half-precision (FP16)*: For the half-precision mode, we store all the data and conduct all the computation in **FP16**. This mode promises the fastest computation, but the numerical errors in this mode are the most severe compared to the other modes.

*Mixed precision (Mixed)*: We also explore a **mixed-precision** mode that uses **FP16** for storage and computation similar to the **FP16** mode; however, it benefits from performing **precalculation** in higher precision using **FP32** arithmetic. This combination is promising for achieving results with higher accuracies while maintaining the performance benefits of half-precision computation since the run time of **precalculation** kernel is a negligible portion of time to compute tiles, and therefore, executing this kernel with higher precision does not result in tangible overheads.

*Half-precision with improved arithmetics (FP16C)*: Finally, we explore another variation of half-precision computation, which again exploits a higher precision mode in **precalculation** with an improved variation of arithmetic that uses Kahan’s compensated summation [Kah65] in **precalculation**. The rest of the steps use **FP16** similar to mixed- and half-precision scenarios. With this compensated summation, we prevent the error propagation from severe cancellations that arise in the **precalculation** in **FP16** mode. Despite the additional computation to compensate for errors in summations, it does not result in significant overheads as, again, the **precalculation** contributes a negligible amount to the time spent to compute a tile and, therefore, contributes very negligible to overall run time. Compared to the Mixed mode, this variation also promises similar accuracy and performance benefits.



## 5.4 Implementation on NVIDIA GPUs

We use C++ for our implementation<sup>2</sup> and aim for NVIDIA GPUs, in particular, NVIDIA V100, NVIDIA A100, and NVIDIA H100, as our target devices (see details for hardware specification of these GPUs in Appendix A5). However, (MP)<sup>N</sup>-GPU approach can be realized on other vendors' GPUs as well. However, source-to-source translation tools might not be able to completely port our implementation since, in some cases, we rely on NVIDIA-specific API and functionalities.

For the experiments, we use GCC compiler v.8.0, together with CUDA v.11.2, which supports all the required functionalities we need to realize (MP)<sup>N</sup>-GPU.

We choose our GPU implementation of Bitonic sort in the sorting step over the popular libraries, such as CUB [NVI23b] or Modern GPU [NC23], as it provides much higher performance for running many sort operations in parallel as needed by (MP)<sup>N</sup>-GPU. For the same reason, we customize our prefix average implementation instead of relying on CUB. We exploit NVIDIA's Cooperative Groups API [NVI23a] for creating the coarse-grained synchronization among the threads for the implementation of the bitonic sort and prefix average operations and exploit GPU's *shared memory* for storage.

For grid-stride loops, we use the kernel launch configuration that matches the hardware architecture: on NVIDIA V100, we use 64 as grid size and 2560 as block size; on NVIDIA A100, we also use 64 as grid size but use 3456 as the block size. Experiments validate that these configurations provide the best performance.

We rely on the Stream Management API [NVI23d] in CUDA for implicit synchronization (especially in the multi-tile code): all the data transfers and kernel executions rely on CUDA streams. We use a maximum of 16 non-blocking streams on one GPU to avoid running into memory consumption limits while keeping a high concurrency.

For FP64 and FP32, we only adopt the data format and simply use the same native mathematical operators in C++. However, FP16, Mixed, and FP16C use the `__half` data type and corresponding intrinsics from the CUDA Math API [NVI23c], as there are no native half-precision data types and operators in C++.

## 5.5 Practical Approach in Assessing Accuracy

Evaluation of reduced- and mixed-precision modes in (MP)<sup>N</sup>-GPU is a central piece of this work, and therefore, we need to establish an accuracy assessment strategy. While in classical HPC research and development, numerical accuracy is often considered, we also consider the trends in the machine learning and data analytics world. We also consider looking into what we call practical accuracy, i.e., to evaluate what the accuracy of a given task (i.e., motif discovery) is when conducting matrix profile analysis using the reduced- and mixed-precision modes. While in many cases, strict numerical accuracy is necessary, in many applications, this is not the case, and having sufficient practical accuracy could satisfy the needs (where we can successfully detect patterns and develop accurate classifiers), despite the presence of numerical inaccuracies. We can argue the

<sup>2</sup>See Appendix A4 for pointers to the code.



benefit of practical accuracy from the following angle: in the case that the repeating patterns within a time series are perfectly identical, matching the patterns to any of the identical patterns is sufficient. In this case, mathematical accuracy is overly strict. While this argument applies to a very specific case, we cautiously state that the implications can be general: application-specific accuracy (or simply practical accuracy) metrics can better quantify the practicality of using reduced precision arithmetics compared to the strict mathematical accuracy metrics.

We use the following accuracy metrics: the first set of accuracy metrics represents the numerical accuracy of results, where to assess accuracy, we compare the numerical differences (or better said, closeness) of results computed via an implementation, e.g., computed with a reduced- or **mixed-precision** mode, to the **CPU**-based reference:

- Relative accuracy ( $\mathcal{A}$ ) [ZYZ<sup>+</sup>18]: the relative discrepancy between the matrix profile computed with a reduced- or **mixed-precision** scheme and the reference **FP64** calculation is considered as the relative error denoted as  $\mathcal{E}$ , where  $0 \leq \mathcal{E} \leq 1$ . We define  $\mathcal{A} = 1 - \mathcal{E}$  as the relative accuracy, measured and reported in percentage.
- Recall rate ( $\mathcal{R}$ ) [CHP21, XB16]: we consider the ratio of the number of matching (*signum*) matrix profile indices between the results computed by the approximate method ( $I_i^T$ ) and exact computation ( $I_i^E$ ) divided by the total number of indices ( $N - m + 1$ ) as *recall rate*.

$$\mathcal{R} = \sum_i \text{signum}(|I_i^E - I_i^T|) / (d \times (n - m + 1)) \quad (5.1)$$

The second set of accuracy metrics provides the means for a practical accuracy evaluation. We specifically consider a specific use case of the matrix profile, e.g., motif discovery or nearest neighbor-based classification of time series data. In these cases, we focus on the actual accuracy in detecting target patterns or classification accuracy.

- Recall for embedded motif detection ( $\mathcal{R}_{\text{practical}}$ ) [ZYZ<sup>+</sup>18]: for pattern detection cases, we quantify the number of successful matches of a specific pattern retrieved by the matrix profile, e.g., when computed using our implementations via a reduced- or **mixed-precision** mode.
- F-score for classification ( $\mathcal{F}_{\text{classification}}$ ) [Tha20]: for classification, we further look at the overall accuracy, F-score, of the classification when we use matrix profile indices computed using our implementations via a reduced- or **mixed-precision** mode. The F-score is the harmonic mean of precision and recall accuracy metrics.

## 5.6 Evaluation

We investigate *accuracy* and *performance* aspects and their trade-offs in various precision modes.

The experiments are mainly conducted on a **DGX-1** at Leibniz Supercomputing Centre [Cen23] with NVIDIA **NVIDIA V100GPU**s and nodes of the Raven supercomputer

at Max Planck Computing and Data Facility [Max23], which includes NVIDIA Tesla NVIDIA A100 GPUs. For the experiments, the configurations to launch all the kernels are tuned to match the corresponding GPU hardware architectures: specifically, we launch 163,840 threads on NVIDIA V100 (80 SMs · 64 warps · 32 threads), and 221,184 threads on NVIDIA A100 (108 SMs · 64 warps · 32 threads).

We use a synthetic dataset of multi-dimensional time series with 80 groups of parameter settings (different  $n$ ,  $m$ , and  $d$ ) for our performance and accuracy evaluation. We observe that the execution time does not vary when based on the employed dataset and stays consistent regardless of individual datasets; therefore, using synthetic datasets is suitable as it allows us full control of the parameters. This dataset includes random noise combined with injected repeating patterns at random locations, providing a reliable basis for pattern detection when we use practical accuracy analysis. See more details of these patterns in the work of Ju et al. [JRY<sup>+</sup>22].

To ensure the stability and reproducibility of the experiments, we repeat each experiment five times and analyze the arithmetic mean of the reported accuracy and performance metrics. In fact, experiments show that (MP)<sup>N</sup>-GPU has stable numerical performance and accuracy for every choice of the platform tested (NVIDIA V100, NVIDIA A100, or H00) and for every tested precision mode. Therefore, we only report the averages of measured metrics in this chapter.

### 5.6.1 Performance Evaluation

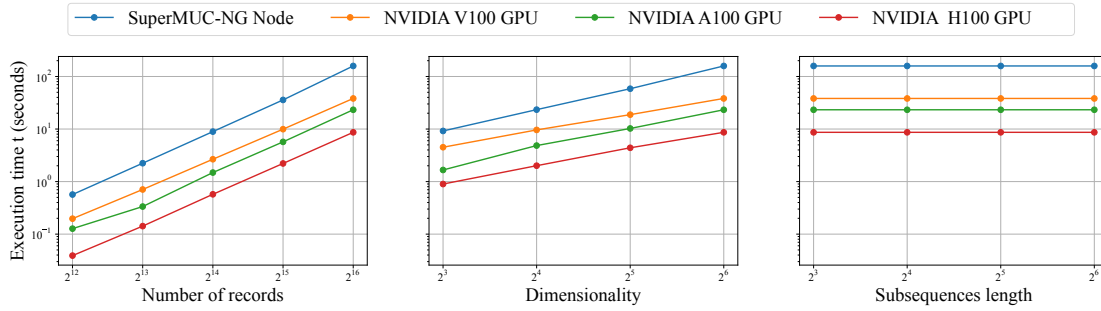
We conduct a number of experiments to evaluate the performance and accuracy of (MP)<sup>N</sup>-GPU in order to answer the proposed research questions at the beginning of this chapter.

#### 5.6.1.1 Performance Overview

We start with examining the performance of (MP)<sup>N</sup>-GPU across different NVIDIA GPU platforms and consider various generations of NVIDIA GPUs with Tesla architectures for this purpose. These include NVIDIA V100 and NVIDIA A100 GPUs, as well as a NVIDIA H100 GPU. We also compare the performance of the (MP)<sup>N</sup>-GPU code against the SuperMUC-NG system. We conduct these experiments with FP64 mode to ensure fair comparison among different implementations since not all of the implementations support reduced-precision arithmetics.

We explore various configurations and parameters (e.g., number of records, dimensionality, and subsequence length), and discuss the *total* execution time as the performance metric. The results of these experiments are summarized in Figure 5.4.

The first observation from Figure 5.4 is that configurations overall, (MP)<sup>N</sup>-GPU can achieve about 3.9x, 6.5x, and 17.0x performance boost in double precision on a single NVIDIA V100, NVIDIA A100, and NVIDIA H100 GPU cards, respectively, in comparison to a single node of the SuperMUC-NG.



**Figure 5.4:** Performance of multi-tile implementation with one tile across different generations of NVIDIA GPUs in comparison to the CPU-based  $(MP)^N$ .

Regardless of the explored size parameter settings, the GPU code provides faster execution compared to CPU implementation. This statement is valid for the GPUs and CPUs of roughly the same generations (ages).

The next observation is the performance behavior (characteristics) of the CPU and GPU codes on different platforms, based on the angle of the lines on the log-log plots in Figure 5.4: the execution time scales quadratically with the number of records (left plot in Figure 5.4) and scales almost linear with dimensionality (in theory, the complexity of the bitonic sort is  $d \times \log^2(d)$  – middle plot in Figure 5.4). Moreover, the execution time is independent of the subsequences' length (Figure 5.4 right).

Overall, the performance characteristics of the CPU and GPU codes are similar, preserving the characteristics in the family of matrix profile STOMP-based algorithms.

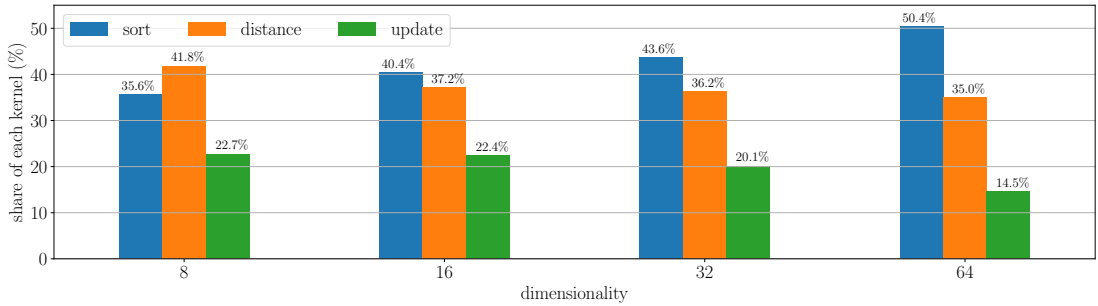
Next, we look at the significance of each kernel based on its contribution to the total run time of  $(MP)^N$ -GPU.

Recapping Figure 5.4, we observe that both the number of records and dimensionality in the input series have a direct influence on the total run time of  $(MP)^N$ -GPU. Therefore, we can assume that kernels that the distance computation (mainly influenced by the number of the records) and sorting kernels (mainly influenced by the dimensionality), might have different significance depending on the different configurations (e.g., different number of the records and dimensionality). Therefore, in this experiment, we set the number of records to a fixed value and investigate the significance of different kernels as we increase the dimensionality.

Figure 5.5 demonstrates the results of this experiment. The first observation on this plot is that for all the presented cases, the shares of distance computation kernel and sorting kernel are fairly close as opposed to the CPU implementation where the run time is highly dominated by sorting kernels (cf. Figure 4.7).

Next, as expected, we observe that the significance of the distance computation kernel (`streaming_dot_product`) is higher in low-dimensional cases as opposed to the high-

## 5 Time Series Similarity Mining on GPUs with Reduced- and Mixed-Precision



**Figure 5.5:** Kernel execution breakdown on NVIDIA A100 GPU.

dimensional cases where the performance starts to be dominated by the sorting kernels (`sort_and_prefix_average`). The reason for this behavior can be traced back to the synchronization overhead in `sort_and_prefix_average`, which increases with dimensionality, causing a slowdown in large dimensionality cases. However, it still outperforms alternative sorting approaches due to the high utilization of the GPU hardware. This argument is based on our previous work on the investigation of a primary version of (MP)<sup>N</sup>-GPU for different implementations of the sorting kernels [JRS21].

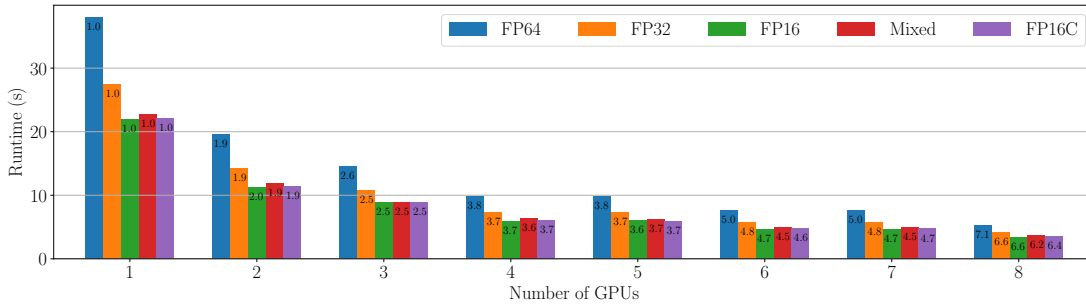
We also look into the GPU utilization on an NVIDIA A100 GPU, and for that, we again rely on NVIDIA Nsight Compute. All of the kernels running on the GPU show memory-bound performance. For the large tiles (i.e.,  $n \geq 2^{16}$ ), `streaming_dot_product` and `update_matrix_profile` use over 80% DRAM and around 70% L2 cache bandwidth. `sort_and_prefix_average` uses over 80% L1/TEX cache bandwidth and around 70% occupancy.

Similar to the CPU implementation, the performance of (MP)<sup>N</sup>-GPU is also bound by memory bandwidth.

### 5.6.1.2 Reduced- and Mixed-Precision Performance

In the next experiment, we compare the performance of various reduced- and mixed-precision modes and investigate how (MP)<sup>N</sup>-GPU in various precision modes scales on a multi-GPU platform.

For this purpose, we run a set of experiments on the DGX-1 system precision mode as parameters. Figure 5.6 shows the results of this experiment. As expected, lower precision modes show higher performance. However, the performance improvement does not scale linearly with the number of bits used in corresponding data types. For instance, going from FP64 to FP32 and from FP64 to FP16, the speedup are only 1.4x and 1.7x, respectively. Looking at the details of the performance of each individual kernel, we notice that all kernels scale almost linearly with the data type as expected, except for `sort_and_prefix_average`, whose performance is mainly dominated by repeating synchronization overheads. Additionally, our implementation uses manual intrinsics and



**Figure 5.6:** Execution time and efficiency of multi-tile implementations with 16 tiles on DGX-1 ( $n=2^{16}$ ,  $d=2^8$ ).

lacks the benefits of intensive automatic compiler optimizations. Finally, the performance improvement for FP16, Mixed, and FP16C modes is similar, as their performance difference that is mainly in precalculation is negligible.

The performance improvement in various reduced- and mixed-precision modes on GPUs is limited to 1.7, while it can be, in theory, as large as 4.0 for FP16. This limit is mainly due to the synchronizations within the bitonic sort kernels.

Resource utilization in FP32 modes is similar to FP64: in this mode `streaming_dot_product` uses around 60% DRAM bandwidth and `update_matrix_profile` uses around 70% DRAM bandwidth and L2 cache bandwidth. `sort_and_prefix_average` uses around 40% L1/TEX cache bandwidth and around 70% compute (SM).

The hardware utilization in other reduced- and mixed-precision modes (FP16, Mixed, and FP16C) is similar to each other: `streaming_dot_product` uses around 30% and `update_matrix_profile` uses over 50% DRAM and L2 cache bandwidth. `sort_and_prefix_average` uses around 20% L1/TEX Cache bandwidth and around 70% occupancy.

Similar to the FP64 mode, the performance in reduced- and mixed-precision modes stays memory-bound.

### 5.6.1.3 Scalability

On the same plot (Figure 5.6), we also look at the scalability of (MP)<sup>N</sup>-GPU on NVIDIA A100 for various precision modes by the number of GPUs as the parameter. By looking at the speedup values for different configurations, which are annotated on the bars in Figure 5.6, we observe the linear scalability, although we also see inefficiencies when using odd numbers of GPUs. We explain these inefficiencies based on the problem setup: since in all these experiments, 16 tiles are used, maximum load balancing among the GPUs and, therefore, perfect scaling is only possible when the number of GPUs is a factor of the number of tiles. This inefficiency can be mitigated by increasing the number of tiles assigned to all GPUs (i.e., reducing tile size). This comes with the potential extra

overhead of tiling when the number of tiles gets large (we will elaborate on this in the following subsections).

When one, two, four, and eight GPUs are used in double precision, our implementation reaches over 88% parallel efficiency (speedup of 7.1). Also, the parallel efficiency stays relatively high for the reduced- and mixed-precision modes (between 78% and 83%).

(MP)<sup>N</sup>-GPU has an acceptable speedup and efficiency when deployed to multi-GPU systems.

### 5.6.2 Accuracy Evaluation

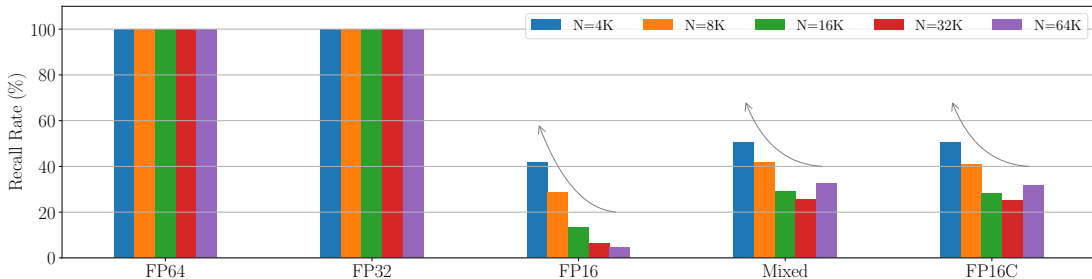
Next, we analyze the *numerical accuracy* and *practical accuracy* of multi-dimensional matrix profile computation on GPUs using (MP)<sup>N</sup>-GPU according to the metrics introduced in Section 5.5.

### 5.6.3 Numerical Accuracy

We run experiments on a single NVIDIA A100 card with various precision modes. As we elaborate in the following paragraphs, the number of records directly impacts the numerical accuracy; therefore, we also focus on the number of records as the main parameter in this experiment. A more detailed analysis of the influence of other parameters (dimensionality and window size) on the numerical accuracy is reported in the work of Ju et al. [JRY<sup>+</sup>22].

Figure 5.7 summarizes the results of this experiment. The FP64 mode on the GPU can generate results identical to the CPU-based implementation. The FP32 mode also results in a high accuracy of roughly 100%. Mixed and FP16C modes result in almost the same accuracy compared to each other and twice the accuracy compared to FP16 mode. However, the numerical accuracy in all the reduced- and mixed-precision modes lies below 60% for all the presented configurations, which is fairly poor.

As the FP16, Mixed, and FP16C modes cannot generate identical results compared to the CPU-based code in the sense of numerical accuracy, we trace the inaccuracies in



**Figure 5.7:** Numerical accuracy (Here only recall rate  $\mathcal{R}$ ) for the single-tile configuration of (MP)<sup>N</sup>-GPU in processing synthetic datasets compared to the CPU-based implementation.

the results to limited numerical accuracy of the half-precision for storage and arithmetic. We take a closer look at the sources of these inaccuracies and attempt to mitigate them.

We have a look at the propagation of errors in streaming dot products. We can describe the iterative computation of  $QT$  in Equation 4.6 over all the iterations (Line 3 in Pseudocode 3) as a large dot product and analyze its sensitivity to rounding errors, based on the analysis provided by Yang et al. [YFS21]. In this analysis, the error bounds for dot-product are proportional to the length of input vectors and machine precision:  $e \propto (n \times \varepsilon)$ . Therefore, by relying on this simple analysis, we can trace the numerical inaccuracies in matrix profile computation in reduced- and mixed-precision modes to *machine error* in floating point format and *tile size*.

- *Machine error* ( $\varepsilon$ ): the single precision ( $\varepsilon_{32} = 2^{-23}$ ) and the half-precision ( $\varepsilon_{16} = 2^{-10}$ ) provide less accurate computation compared to the double precision ( $\varepsilon_{64} = 2^{-52}$ ). Since distance computation in matrix profile computation is performed in a *z-normalized* domain, the distance values often have a limited dynamic range. Therefore, accurate computation of the difference in distance of subsequence pairs often comes down to having a small  $\varepsilon$ .
- *Tile size*: when employing the streaming dot product formulation, round-off error in the computation of the distance matrix grows drastically with the size of the tiles used. This effect is also visible in Figure 5.7. We observe that when reducing the number of records in the time series (here, the number of records and tile size are identical as we only have one tile), the accuracy of reduced and mixed-precision modes increases (direction of arrows annotated on the bars).

Using smaller tiles can reduce the propagation of the numerical error for computing the matrix profile with reduced- and mixed-precision.

This property can be combined with the tiling scheme in Pseudocode 6 to improve the accuracy in reduced- and mixed-precision computation modes.

#### 5.6.4 Practical Accuracy

So far, we only focused on the evaluation of the accuracy of (MP)<sup>N</sup>-GPU in the sense of the mathematical difference between the approximate solution computed with (MP)<sup>N</sup>-GPU and a canonical solution, i.e., computed on CPUs. However, as we discussed in Section 5.5, in many applications, the practical accuracy can be sufficient.

We, therefore, focus on two illustrative examples to show the accuracy of (MP)<sup>N</sup>-GPU in practice despite the mathematical inaccuracies. In the first example, we rely on the accuracy for the detection of injected patterns, and in the second one, we rely on the accuracy of classification.

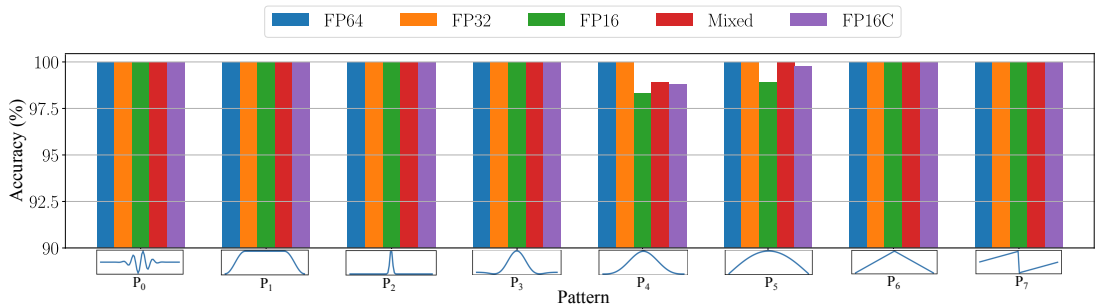
We build the pattern detection example on top of the evaluation scheme proposed and used by Zhu et al. [ZYZ<sup>+</sup>18]: we consider a predefined set of primitive shapes and embed a predefined number of these primitives onto predefined random locations



and dimensions. The practical accuracy evaluation quantifies the number of matching primitives that are successfully derived from a matrix profile analysis, i.e., a recall value  $\mathcal{R}_{practical}$ . For instance, in the simplest case, we consider a set of only one primitive, e.g., a sine pulse. Now let's say we have a two-dimensional time series generated from white noise, and we simply inject two of these sine pulses ( $P_0$  and  $P_1$ ) onto the two dimensions and locate the start of these pulses at random locations, e.g.,  $T_0$  and  $T_1$ . This would generate a two-dimensional time series, with a matching pair of one-dimensional motifs located at  $T_0$  and  $T_1$ . Now let's assume that the canonical matrix profile computation (i.e., on CPU with full precision) can successfully determine that the matching pair for  $P_0$  is  $P_1$  and vice versa (both of them), which is most probably the case (that is, the whole point of a matrix profile analysis in this scenario anyway). In the perfect case where a reduced- or mixed-precision mode also detects the correct matches for both  $P_0$  and  $P_1$ , it would have 100% accuracy ( $\mathcal{R}_{practical} = 100\%$ ) compared to the canonical matrix profile analysis with full precision.

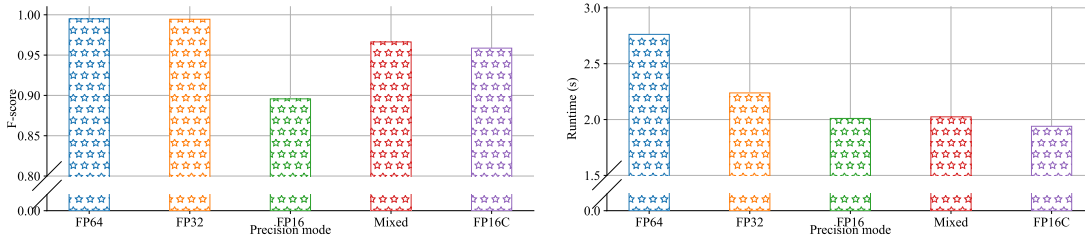
Now let's describe the concrete scenario: we use a set of eight primitive patterns for embedding, which are illustrated and marked with  $P_0 - P_7$  in Figure 5.8. We inject 100 instances of one of these patterns (subsequence length is set to 64) into a random multi-dimensional time series with 64K records and 64 dimensions at random locations. With this setup, we compute the matrix profile of the canonical on CPU, which results in 100% accurate detection of matching patterns for all of  $P_0 - P_7$ . Now we look at the reduced- and mixed-precision computation modes: With the exception of  $P_4$  and  $P_5$ , where only 98% accuracy is achieved when employing FP16, Mixed, and FP16C modes, all the reduced- and mixed-precision modes can achieve exact 100% practical accuracy, i.e., they detect *all* injected patterns.

While this pattern embedding example is a simplified scenario, and in the real world, the patterns, their shapes, and their embedding structure (e.g., in different locations and dimensions) can be more complicated, we still consider this an illustrative example to show: more exhaustive evaluation of practical accuracy for pattern detection goes beyond the scope of this thesis, and we limit to this analysis.



**Figure 5.8:** Practical accuracy ( $\mathcal{R}_{practical}$ ) of single-tile implementation for pattern detection. We plot the patterns with time as x-axis ( $x \in [0, m)$ ) and the normalized values as y-axis ( $y \in [-1, 1)$ ).





**Figure 5.9:** Accuracy ( $\mathcal{F}_{classification}$ ) and performance of the nearest neighbor classifier with respect to various precision modes on the [HPC-ODA](#) dataset.

In the second practical example, we look at the task of identifying (classifying) benchmarks from operational data gathered from real-world monitoring of benchmark runs on an [HPC](#) system. In this example, we employ a nearest neighbors classifier to the matrix profile index and compare the classification accuracy in practice when the matrix profile is computed with full or reduced precision arithmetics.

We exploit the *Application Classification* segment of a public [HPC-ODA](#) dataset [[Net20](#)] (see Chapter 2 for more details on this dataset). This dataset includes labeled performance data collected while running different benchmarks ([HPL](#), [AMG](#), etc.) on 16 compute nodes for one day with one [HZ](#) sampling rate. We select 16 distinct sensors (performance metrics) on different nodes (e.g., cache miss rates, branch instructions) for the multi-dimensional matrix profile analysis and split the dataset along time into two portions, a reference set and a query set, each of which includes continuous operational data for half a day. We conduct the matrix profile analysis with different precision modes and build a simple classical nearest neighbor *classifier* on top of the matrix profile index: the classifier identifies the matching subsequences using the matrix profile index and uses the labels of the segments in the reference set to determine the application class of the segments in the query set. In essence, this simple model only requires the computation of the matrix profile (index) to work, and we can consider the matrix profile computation as its training step.

As shown in Figure 5.9 left, while the accuracy of the classifier is reduced slightly with reduced- and [mixed-precision](#) modes, for the [Mixed](#) and [FP16C](#) modes, it is still over 95%, and even [FP16](#) also reaches almost 90%. In Figure 5.9 right, a slight performance increase is visible when exploiting reduced- and [mixed-precision](#) modes despite the small size of the [HPC-ODA](#) dataset. The reason for this minimal performance improvement is that the [HPC-ODA](#) is relatively small, and therefore, the performance advantages of reduced- and [mixed-precision](#) computation modes that we discussed before do not kick in fully here.

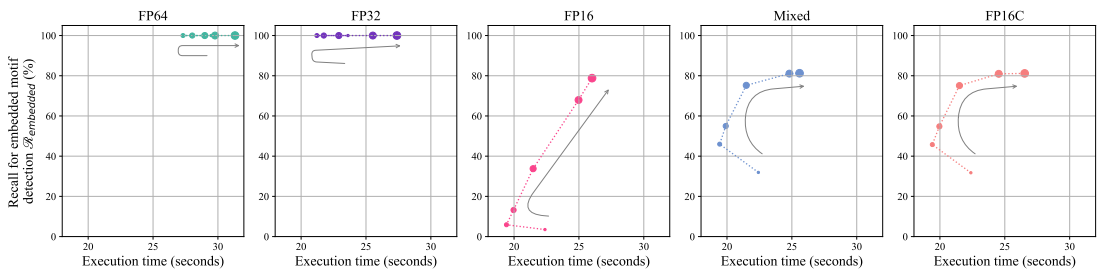
Despite the numerical inaccuracies (i.e., in the sense of mathematical inaccuracy), (MP)<sup>N</sup>-GPU, with reduced- and [mixed-precision](#), can deliver precise pattern detection and classification in practice (i.e., in the sense of practical accuracy).

### 5.6.5 Accuracy-Performance trade-off

In Section 5.6.2, we showed that the mathematical accuracy relates to the tile size. In particular, increasing the tile size can lead to less accurate results. Additionally, the tiling scheme can be used as an accuracy mitigation scheme. The idea is to use more but smaller tiles on the GPUs. Also, the final merging of tiles in the current implementation of  $(MP)^N$ -GPU is executed by the CPU (see Pseudocode 6), which results in an overhead increase when the number of tiles increases. This implementation (i.e., with merging on CPU) would potentially also have extra overhead in the tiling mechanism: as the number of tiles increases, more data movement rounds by the tiling scheme are performed. Also, as tile sizes are reduced, the amount of work per tile and, thus, the time to compute a tile is reduced. Although data movements are executed asynchronously to the kernel, movement latencies and overheads can potentially show up as the tiles get smaller.

In the final experiment in this chapter, we take a look at these performance-accuracy trade-offs. We conduct an experiment to study the Accuracy-Performance trade-off (Figure 5.10) by increasing the number of tiles from 1 to 1,024 (and reducing the tile size accordingly).

Using more tiles increases the numerical accuracy for FP16, Mixed, and FP16C modes. This experiment is also in accordance with the results presented in Figure 5.7 and the following discussion regarding the direct relation between the error propagation and tile sizes. At the same time, ignoring the initial performance boost from 1 tile to 256 tiles that is due to the concurrency invoked by using multiple streams in the implicit synchronization scheme (i.e., CUDA streams), increasing the number of tiles has a slight negative impact on the execution time (as expected). However, in a special configuration, using 256 tiles allows FP16, Mixed, and FP16C modes to have 2x accuracy improvement and an even shorter total execution time compared to a one-tile run. With this configuration, Mixed and FP16C modes reach almost 80% accuracy with 256 tiles. Also note that the overhead of the merging step is negligible for large problems, leading us to an overall more efficient configuration.



**Figure 5.10:** Accuracy-Performance trade-offs of multi-tile implementations on one NVIDIA A100 GPU with increasing number of tiles. ( $n=2^{16}$ ,  $d=2^6$ ,  $m=2^6$ ). The size of the markers indicates the number of tiles  $n_{tile}$ . We annotate arrows next to the data to indicate the direction of the increase in the number of tiles.

Overall, using more tiles is a plausible setting due to the accuracy boost and the insignificant performance drop.

## 5.7 Summary

In this chapter, we presented a GPU-based approach to accelerate multi-dimensional matrix profile computation. We discussed the design and implementation of this approach and introduced various reduced- and mixed-precision modes for multi-dimensional matrix profile computation. We conducted extensive accuracy and performance measurement experiments and demonstrated performance, characteristics, accuracy of reduced and mixed-precision computation schemes, and their performance-accuracy trade-off.

While other data formats, such as TF32 and BFLOAT16, might sound interesting to investigate (we also extended (MP)<sup>N</sup>-GPU with BFLOAT16), our further investigations (not reported in this work) suggest that these formats are not beneficial for matrix profile computation. Relying on the discussion in Section 5.6.2, these formats do not provide a wider mantissa, and the wider dynamic range compared to FP16 in these formats is not often helpful in multi-dimensional matrix profile computation. At least they do not make any improvement to the results of the experiments we presented in this chapter.

Overall, the solution presented in this chapter (multi-GPU + reduced precision arithmetics + tiling) demonstrates efficient and accurate data mining with reduced precision arithmetics. This solution is a significant step towards more efficient HPDA for time series analysis and pushes the limits of state-of-the-art significantly.

## 6 Other Tackling Point; Algorithmic Approach for High-Performance Similarity Mining

The approach described in this chapter was previously published in [ISC'23], where the author of this thesis made major contributions to the contents, including the development of the prototype codes, and evaluations.

### 6.1 Motivation

In the previous chapter, we focused on  $(MP)^N$  and  $(MP)^N$ -GPU to enable time series mining using matrix profiles on modern HPC systems with high performance. The approaches we discussed in the chapters (i.e.,  $(MP)^N$  and  $(MP)^N$ -GPU) are built on top of the existing well-established methods for matrix profile computation, e.g., STOMP and mSTAMP. In particular, they mainly use streaming dot product formulation [Nes23] at their core. These approaches mainly focus on scalability and performance improvements using well-established HPC methods and techniques.

However, the HPC community is well aware of the critical role that software innovations and algorithmic improvements play in moving HPC applications forward while settling in the exascale era [Pod]. This highlights the importance of using suitable parallel algorithms and tools on large-scale HPC systems to meet the processing speed, efficiency, and memory requirements, e.g., in matrix profile computation, for *large* datasets as they appear in real-world use cases. This importance also drove us to revisit the problem of computing the matrix profile on HPC systems from a different angle. We already discussed in previous chapters (e.g., in Chapter 2) that matrix profile computation, in essence, is nothing different from solving the nearest neighbors problem associated with the subsequence similarities (i.e.,  $T^r \bowtie_{1nn}^k T^q$ , see Definition 15). The nearest neighbors computation problem has been around for a while and has been visited by the HPC community in various works. This includes various approaches for addressing large-scale nearest-neighbor problems and approximate tree-based methods (see Chapter 2). To the best of our knowledge, no research in the literature focuses on investigating and applying such approaches and techniques to matrix profile computation. In particular, no research has been conducted to employ data structures like trees to prune the search space associated with the nearest neighbor search operations during matrix profile computation to reduce computational costs and to target large-scale problems on HPC systems.

In this chapter, we take one step back and look at the matrix profile computation in a single-dimensional case to establish a base for employing existing methods and techniques in the HPC community for computing the matrix profile. With this, we move away from the streaming dot product, i.e., STOMP-based formulations that have both benefits and shortcomings: specifically, we observe in the last chapter that the streaming dot product formulation is the source of the presence of error propagation in reduced and mixed-precision computation schemes. However, we are also aware that this formulation reduces the computational costs significantly. Therefore, as a first step in the investigation of alternative methods, we leverage an approximate tree-based method for the computation of matrix profiles to look into the performance benefits and performance-accuracy trade-offs for such alternatives.

The state-of-the-art STOMP-based methods for computing the *nearest-neighbors* problem associated with matrix profile computation are based on exhaustive search operations [YZU<sup>+</sup>16, YZU<sup>+</sup>16, HHvW<sup>+</sup>20]. In these approaches, the computational costs scale quadratically with the size of the datasets (i.e., the number of records). These methods typically try to mitigate these computational costs by extensive arithmetic optimizations of compute kernels [ZYZ<sup>+</sup>18], the use of accelerators in computation [ZZS<sup>+</sup>18, JRY<sup>+</sup>22], and deployment on a cloud-based [ZKS<sup>+</sup>19] or an HPC system [RKY<sup>+</sup>20, Pfe19]. However, these exact solutions are generally inefficient for *large* datasets due to the quadratic scaling of computational costs. On the other hand, approximate approaches [ZYZ<sup>+</sup>18, SZSF<sup>+</sup>19] are drawing increasing attention as they can provide solutions that are much more efficient to compute while being also accurate enough in practice. SCRIMP++ [ZYZ<sup>+</sup>18] relies on estimating the solution by computing the matrix profile index on a subset of the input dataset and refining the solution. LAMP [SZSF<sup>+</sup>19] computes a prediction of matrix profile index using a pre-trained deep neural network model. DAMP [LWM<sup>+</sup>22] is a special solution designed for the computation of time series *discords*. All these approaches [ZYZ<sup>+</sup>18, SZSF<sup>+</sup>19, LWM<sup>+</sup>22] still suffer from various limitations, including excessive computational costs, being restricted to specific application scenarios, or lacking a suitable parallelization scheme.

In this chapter, we build and expand on the family of iterative approximate nearest-neighbors [Cla83] algorithms relying on tree data structures and randomized schemes [Ben75, AMN<sup>+</sup>98] to prune the search space and reduce computational costs. This allows for an approximate solution that its computational costs scale sub-quadratically with the size of the input time series while keeping the general applicability to different application scenarios of matrix profile methods. We address computation for *large datasets*, target large-scale CPU-based HPC systems, and use weak scaling as the relevant scenario to scale to larger datasets. We demonstrate that, when applied to matrix profile computation in weak scaling, the state-of-the-art nearest neighbor approach [XB16] suffers from massive communication overheads. We address these scaling challenges arising from these communication overheads as the core contribution in this chapter.

We specifically introduce two optimizations to address scaling challenges: 1) we stager the stages of the nearest neighbor computation algorithm in consecutive iterations in a pipeline fashion to overlap the construction of the parallel tree data structure and the

follow-on search operations; and 2) we enable control over the granularity of parallelism by constructing a forest of parallel trees on smaller ensembles of resources.

Overall, our tree-based solution enables us to exploit and explore the logarithmic nature of tree-based nearest-neighbor schemes for the first time in matrix profile computation. Combined with the scaling so optimizations, we achieve a scalable solution capable of solving large-scale real-world use cases on HPC systems.

We therefore consider the following research questions (RQs) to structure the discussions in the rest of this chapter.

## 6.2 Research Questions

- RQ: can we quantify the benefits and trade-offs for the tree-based approach in comparison to the existing approximate approaches?
- RQ: how does the tree-based approach scale on HPC systems, and what are the sources of scaling overheads?
- RQ: how do the proposed optimization mechanisms affect the overheads, and how do they interplay?
- RQ: how does the tree-based approach scale overall?

To answer these questions, we make the following contributions:

1. We develop and apply an iterative tree-based nearest neighbor algorithm for the approximate computation of the matrix profile.
2. We perform a detailed analysis of the performance, accuracy, and scaling behavior of the iterative tree-based approach and demonstrate its benefits.
3. We extend the state-of-the-art iterative nearest neighbor with a combination of a pipelining mechanism and creating a forest of trees to scale the matrix profile computation on the HPC systems.
4. We demonstrate the region of benefit where the tree-based approach is superior to alternatives when applied to real-world datasets and scenarios.

We further demonstrate that, when compared to the prior art, the proposed optimizations improve the scalability of the tree-based matrix profile computation on HPC systems by boosting the parallel efficiency significantly. In particular, when increasing the resources by three orders of magnitude on the SuperMUC-NG system, the prior art suffers drastically from scaling bottleneck, and our optimizations *double* the parallel efficiency for this setting.

We also showcase the performance of our approach for large-scale problems by computing the approximate matrix profile for a time series with one billion records on 48K cores of the system with 99% accuracy in under 20 minutes, which is 3-100 times faster than other alternative approaches.

## 6.3 Taxonomy of Methods for Computing the Matrix Profiles

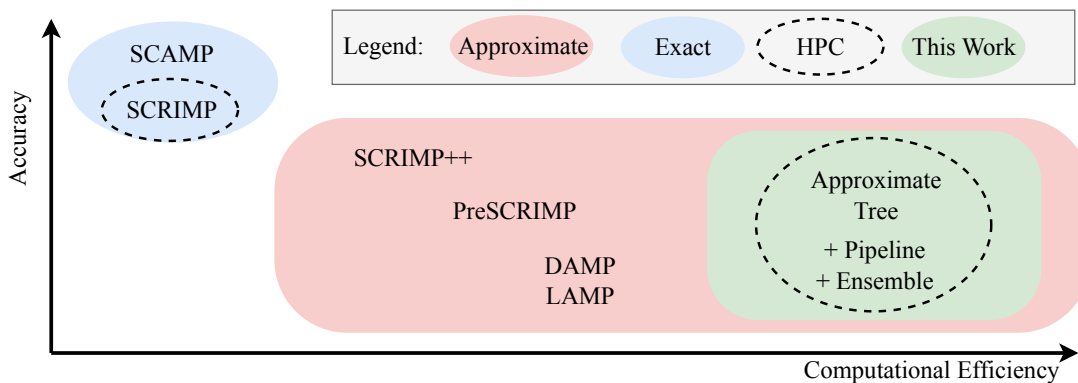
To better understand the relationship between the tree approach we are presenting in this chapter and the existing work in the literature, we provide a taxonomy of methods. We identify four main categories of approaches in the literature that relate to this work: Figure 6.1 illustrates various approaches and visualizes these categories on a schematic accuracy-efficiency trade-off graph.

### 6.3.1 Exact Methods

The state-of-the-art exact method (Blue Region in Fig 6.1) for computing the matrix profile, the *SCAMP* [ZKS<sup>+</sup>19] algorithm, offers  $O(n^2)$  complexity. *SCAMP* exploits GPUs for accelerated computation and targets Cloud environments. Zhu et al. [ZYZ<sup>+</sup>18] propose *SCRIMP*, which is also an exact approach and has similar computational properties to *SCAMP*. This approach is further extended and deployed on HPC systems [Pfe19]. Approximate approaches (such as the tree-based approach) suggest reasonable alternatives to reduce computational costs.

### 6.3.2 Approximate Methods

Zhu et al. propose the approximate methods (Red Region in Fig 6.1), *PreSCRIMP*, and *SCRIMP++* [ZYZ<sup>+</sup>18]. Although *PreSCRIMP* is introduced as a preprocessing step for the *SCRIMP++* algorithm, it stands as a standalone approximate solution as well. *SCRIMP++* itself is both an exact and approximate solution that iteratively refines an initial approximate solution computed using *PreSCRIMP*. However, despite the promising results and properties, none of them accomplish sub-quadratic computational costs. *LAMP* [SZSF<sup>+</sup>19] uses a neural network model to approximate matrix profile indices. Although this approach achieves significant speedups in comparison to exact solutions, it targets computation of matrix profile for data streams in *real-time* settings and, there-



**Figure 6.1:** Overview of various methods representing their trade-off in accuracy vs. computational efficiency



fore, is not ideal for batch processing of large-scale datasets. Moreover, the authors clarify that the accuracy (e.g., false-positive rates) significantly depends on the quality of a reference dataset used to train the model. Finally, Zimmerman et al. [ZKS<sup>+</sup>19] investigate the approximation of a matrix profile using reduced-precision computation and conclude that a single-precision computation suffers from a significant loss of accuracy. Our investigations suggest that this loss mainly stems from the propagation of numerical errors in the streaming dot product formulation used as the core in all existing methods [JRY<sup>+</sup>22]. None of the above-mentioned limitations applies to the tree-based approach.

### 6.3.3 Tree-based Nearest Neighbor Methods

Tree data structures traditionally are used to accelerate ModSim computations in HPC: Barnes-Hut [BH86] and Fast Multipole Methods [Rok85] are examples of such algorithms that have been used to mainly speedup force computations in N-body problems. These algorithms constantly attract attention in other fields, e.g., in the field of data analytics, where, for instance, Van Der Maaten [VDM14] exploits a Barnes-Hut method to accelerate the t-SNE method. The family of nearest-neighbors problems has also benefited from tree data structures for many years [AMN<sup>+</sup>98]. These problems are proven to gain “substantial speedups” [VDM14] by exploiting tree data structures, such as KD-trees [Cur15]. Xiao et al. [XB16] use parallel randomized KD-trees to solve nearest-neighbors problems with *approximation* targeting high-dimensional datasets and show that, in case of low intrinsic dimensionality in the dataset, their approach can solve nearest-neighbors in an exact fashion in *linearithmic* time. Although the dual-tree algorithms are the “fastest known way to perform nearest-neighbor search” [Cur15], their distributed memory support is limited, and studies are restricted to smaller dimensionalities [SHWG15]. Further, a distributed memory approach based on dual-tree methods would result in similar overheads (e.g., overheads in tree construction and not only search overheads), which we address in our work. Overall, the tree approaches (Green Region in Fig 6.1) remain viable to be employed for matrix profile computation.

### 6.3.4 Methods Targeting HPC Systems

Among all the approaches discussed, only SCRIMP is deployed on large-scale HPC systems (Encircled in [dashed] ellipses in Fig 6.1). However, it is still implemented as an exact approach with  $O(n^2)$  computational costs.

Despite the long history, there are limited *nearest neighbor* methods that leverage tree data structures while targeting the deployment on HPC systems: main approaches are FLANN [ML14], PANDA [PSS<sup>+</sup>16], and RKDT [XB16, YHA<sup>+</sup>15]. Among these, RKDT approach [XB16, YHA<sup>+</sup>15] is well-studied on distributed memory systems and works well on large datasets with fairly high dimensionality (which is a requirement in the case of matrix profile computation), and therefore is used as the base of the work presented in this chapter. However, as we discuss later, this approach still suffers from various inefficiencies in the scope of matrix profile computation. In this chapter, we



experimentally demonstrate these inefficiencies and provide algorithmic redesigns and optimizations to enable a more scalable tree-based matrix profile computation on HPC systems.

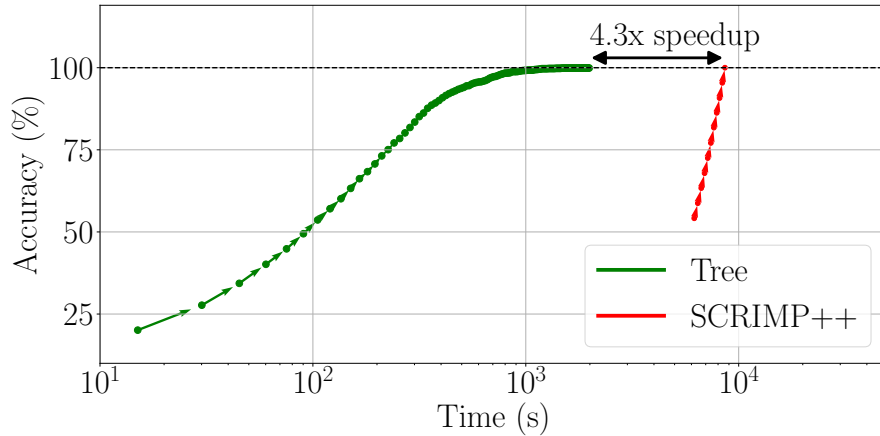
### 6.4 Benefits of Employing Approximation and Trees for Matrix Profile Computation

Computing the matrix profile for two input time series, each including  $n$  records ( $n$ ) require  $O(m \cdot n^2)$  computation corresponding to the cost to compute the distance matrix. However, state-of-the-art methods (e.g., SCRIMP++ [ZYZ<sup>+</sup>18]) achieve this in  $O(n^2)$  by using the *streaming dot product* formulation for distance computation. Tree-based methods cannot benefit from this formulation; however, they can prune parts of the computation of elements in the distance matrix that are less likely to be needed, rendering sub-quadratic costs in  $n$  while keeping the linear cost in  $m$ . Therefore, based on subsequence length ( $m$ ) as a parameter, the tree-based approach has a performance trade-off that is not present in the state-of-the-art exact methods: in STOMP-based methods, the run time does not scale with the subsequence lengths, which is a beneficial feature for the analysis of arbitrarily large sequences and patterns. The tree-based approach trades this property in favor of enabling fast analysis of larger time series. With such a trade-off, this method becomes beneficial for the cases where the number of records  $n$  is large (e.g.,  $n > 1,000,000$ ), and the window size is relatively small (also see the problem settings in the work of Lu et al. [LWM<sup>+</sup>22]). Additionally, problems with large window sizes are not the ideal setting for tree-based approaches, and the efficiency of the tree data structure might be suboptimal due to the curse of dimensionality [JOR11, XB16]: trees are known to degenerate for high dimensional cases and, hence, are less effective for nearest neighbor computations in high dimensional settings. Still, commonly, approximate methods [XB16] remain effective even in relatively high dimensional settings (depending on the properties of datasets), where this dimensionality can grow and reach 100s to 1,000s. This is sufficient to cover many use cases appearing in real-world scenarios. Additionally, we still argue that cases with extremely large window sizes, e.g.,  $m > 10,000$  (e.g., scenarios reported in this work of Zhu et al. [ZYZ<sup>+</sup>18]), are typically over-sampled scenarios and can be downsampled and mapped to problems with much smaller window sizes, where the tree-based approach can still be advantageous. We shed light on this trade-off in practice in this chapter.

Moreover, in the approximate case, only parts of retrieved neighbors match the exact reference segments; an approximate method sacrifices the accuracy of retrieved results in favor of computational efficiency. Consequently, there is a trade-off between the accuracy and computational efficiency of methods (see Figure 6.1 – more details in Section 6.8.2).

#### 6.4.1 Potentials of Tree-based Methods

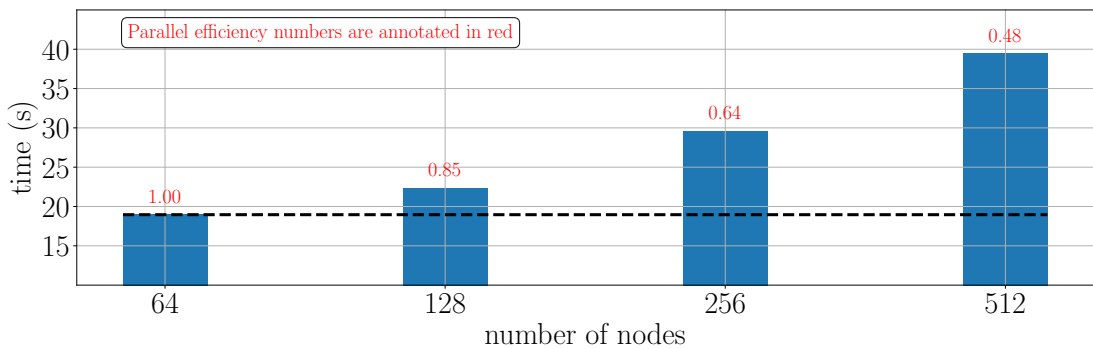
In real-world cases, we are increasingly dealing with larger datasets, and billions of records in these time series are becoming common. In such scenarios, it is increasingly



**Figure 6.2:** Tree-based approach compared to the classical `SCRIMP++` method, both iteratively progressing (single core runs,  $n = 1000K$ ,  $m = 128$ ).

important to provide the community with algorithms that can facilitate the analysis of such a large number of records. Tree-based approaches can be particularly beneficial in these cases and can provide sufficiently accurate *approximate* solutions with much better computational efficiency compared to the state-of-the-art. No method in the literature exploits tree-based nearest-neighbors in the context of matrix profile computations, and our work demonstrates the benefits (see Figure 6.2). In particular, in Figure 6.2, we observe the rapid convergence of a tree-based approach employed for matrix profile computation compared to `SCRIMP++`.

Additionally, the state-of-the-art tree-based method [XB16] suffers from inefficiency in weak scaling when applied to matrix profile computation (See Figure 6.3). Consequently, when exploiting these methods, computing the matrix profile for larger time series becomes inefficient as we increase the size of input datasets along with the number of resources. In our work, we explicitly focus on the scalability issues and provide



**Figure 6.3:** Weak scaling of the forefront tree-based nearest-neighbor method used for matrix profile computation.

optimization methods to improve the scalability of tree-based approaches. With these optimizations, we are able to compute the matrix profiles for large datasets more efficiently.

Nevertheless, as we are targeting large time series, the tree-based method is demanding with respect to compute, memory, and networking resources, and therefore we are targeting HPC systems. However, given accuracy, and computational efficiency, the tree-based approach is highly beneficial and competitive.

## 6.5 Parallel Tree-based Approach

In this section, we move to a parallel setting targeting distributed memory systems. We will regularly use underlined words similar to “parallelization” to highlight when a scheme or term only applies to this setting (i.e., it is only applicable in the case of the parallel scheme). Additionally, we highlight parts in text or Pseudocodes that are affected and mitigated by our optimization schemes in red. Finally, we highlight regions of codes (Phases) in different text blocks with background colors in green, red, and blue. This helps to infer correspondences between the text and the elements in pseudocodes more easily.

However, before discussing the parallelization scheme, we first elaborate on the approach tree approach itself. We consider reference and query time series  $T_R \in \mathbb{R}^r$  and  $T_Q \in \mathbb{R}^q$ , and subsequence length  $m$ .

---

**Pseudocode 7** Computation of matrix profile based on randomized KD-tree *in parallel*.

---

**Input:** The reference and query time series  $T_R$  and  $T_Q$ .

**Output:** The matrix profile  $P$  and index  $I$ .

---

```

1:  $I \leftarrow \{-1\}$ ,  $P \leftarrow \{\infty\}$ 
2:  $R, Q = \text{sliding\_window\_and\_znormalize\_distribute}(T_R, T_Q)$ 
3: for  $i \leftarrow 0$  to  $T$  do in parallel on pipes and on ensembles
4:    $S_0 = \text{random\_direction\_broadcast}()$  ▷ Phase 1 (P1)
5:    $R_0^{\text{rot}}, Q_0^{\text{rot}} = \text{random\_transform\_parallel}(R, Q, S_0)$ 
6:   for  $l \leftarrow 0$  to  $L$  do on ensembles
7:      $\text{med}_1 = \text{max\_var\_quick\_select}(R_1^{\text{rot}}, S_1)$  ▷ Phase 2 (P2)
8:      $S_{l+1}, R_{l+1}^{\text{rot}}, Q_{l+1}^{\text{rot}} = \text{split\_redist\_balance}(R_1^{\text{rot}}, Q_1^{\text{rot}}, \text{med}_1)$ 
9:   end for
10:  for leaf in tree leaves do eagerly
11:     $I^i, P^i = \text{NN\_parallel}(Q_{\text{leaf}}^{\text{rot}}, R_{\text{leaf}}^{\text{rot}})$  ▷ Phase 3 (P3)
12:  end for
13:   $I, P = \text{merge\_iteration\_results\_all2all}(I, I^i, P, P^i)$ 
14: end for

```

---

The first step is to address matrix profile computation with Nearest Neighbor (NN) join formulations. For that, we represent the time series windows (subsequences) as a set of objects for the nearest neighbor join operation. Therefore, we derive two datasets  $R \in \mathbb{R}^{r \times m}$  and  $Q \in \mathbb{R}^{q \times m}$  (corresponding to the reference and query time series, respectively) by sliding windows of size  $m$  on each time series (Line 2 in Pseudocode 7). We include all the generated subsequences in  $R$  and  $Q$  and run the nearest neighbor computation to solve the corresponding nearest neighbor join ( $R \bowtie_{1nn}^m Q$ ). The output of this join operation is the matrix profile (see Chapter 2 for more details).

This computation can be addressed using an *iterative* and *pruned* method instead of directly solving NN ( $R, Q$ ) by computing pairwise distances and streaming dot product formulation. For that, often KD-tree data structures are employed. In Pseudocode 7, we provide a method based on the approach by Xiao et al. [XB16] adopted to compute the matrix profile. In this approach, in each iteration, nearest neighbors are computed at the leaves of the tree in a greedy fashion (Line 11), and the partial iteration results are iteratively updated (merged) to shape the resulting matrix profile and index (Line 13). This iterative process continues for a predefined number of iterations or until a certain accuracy level estimation is reached.

This approach starts with initializing the resulting matrix profile ( $P$  and  $I$ ) to neutral values (Line 1). Next, the two input datasets  $R$  and  $Q$  are computed by applying sliding windows on the input series and z-normalizing each sample (Line 2). In multiple iterations (Line 3), the input datasets ( $R^{rot}$  and  $Q^{rot}$ ) are randomly (actually “reflected” as we employ a householder transformation instead of a rotation matrix) rotated (Phase 1). Specifically, a householder transformation is applied to both the reference and query sets (Lines 4-5). In each iteration, a tree is constructed (Lines 6-8, Phase 2) on top of the rotated datasets, where at each level of the tree (Line 6),  $R^{rot}$  and  $Q^{rot}$  are partitioned (Line 8) into smaller subsets based on the median (Line 7) of the principal direction (i.e., the direction with maximum variance). After building the tree, a pruned nearest neighbor in each leaf of the tree is solved, which exploits BLAS (DGEMM) operations (i.e., greedy search on leaves – Line 10). At the end of each iteration, the partial results of iterations are merged (Line 13, Phase 3) using element-wise minimum and arg-minimum operations (refer to Chapter 2 for a brief background on these operations).

From hereon, we move to a parallel setting where we realize the same scheme proposed by Xiao et al. [XB16] to leverage, e.g., multi-processing and message passing. We improve this approach with extra optimization mechanisms. In this approach, a distributed tree data structure is constructed to collocate the reference and query points with the most similarity in the leaf in each node/process. The reference and query sets ( $Q$  and  $R$ ) are statically partitioned and distributed accordingly among different nodes/processes initially. During the iterations, these sets are repartitioned and shuffled around in parallel to construct the tree, i.e., to bring *similar* reference and query data in the same nodes/processes of the tree (Lines 7 and 8 in parallel). Then, the matrix profile computation is reduced to a series of BLAS operations (DGEMM) in the leaves of the tree running on different nodes/processes in parallel (Line 11).

## 6.6 Scalability Challenges and Overcoming Them

The parallel approach requires static partitioning of reference and query sets (`distribute` in Line 2), `broadcasting` the transformation direction (in Line 4), a series of `reductions` and pair-wise data exchanges as part of distributed approximate `quick_select` method used to compute medians (Line 7). The reference and query sets are split according to the median value and reshuffled (`redistribute` and `balance`) using a series of pair-wise exchanges during the construction of the tree (Line 8). Furthermore, an additional *all-to-all* operation in Line 13 is used to merge iteration results  $P^i$  and  $\bar{I}^i$ . This brings the resulting  $P^i$  and  $\bar{I}^i$  to the right process/node, where the original reference sets are residing according to the static partitioning in the first step, and finally, it merges the results using element-wise comparisons.

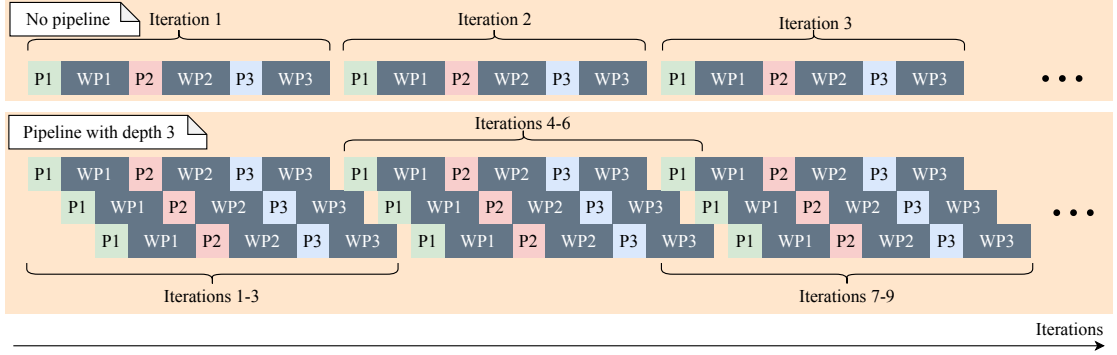
All these mechanisms add extra communication overhead at scale. These overheads under the matrix profile setting running on a large number of nodes are extreme. In such settings, communication overheads dominate the run time, in comparison to the time spent on transformation and `DGEMM` operations that run fully in parallel. In more detail, in large scenarios (at scale), the run time of the *all-to-all* collectives in `merge_iteration_results_all2all` grow drastically and dominate the overall run time. Additionally, the pairwise exchanges in `split_redist_balance`, as well as the reductions in `max_var_quick_select`, are another major scaling bottleneck. For instance, our evaluations show that these overheads combined comprise  $\approx 50\%$  of the run time for large jobs (see Section 6.8 for a detailed analysis of the overheads).

We provide two complementary mechanisms to mitigate these extra overheads:

1. Based on the analysis of these overheads (see Section 6.8), we identify three Phases (see Section 6.5), denoted as P1, P2, and P3, in Pseudocode 7 (highlighted from top to bottom in `green`, `red` and `blue`) and introduce a scheme to stagger the phases of iterations in a **pipelining fashion** (by adjusting `Line 3` in Pseudocode 7). This allows for concurrent and partial execution of multiple iterations, resulting in the overlap of communication and computation of multiple phases and, therefore, hiding the communication latencies of the phases. This approach is, in particular, effective in reducing the latencies of *all-to-all* collectives.
2. Complementary to the pipelining mechanism, we create a **forest of trees on multiple exclusive ensembles** of resources (by adjusting `Line 3` in Pseudocode 7). Overall, this allows to coarsen the granularity of parallelism and therefore reduces the communication overheads. This approach is particularly effective in reducing the communication costs in the series of pairwise exchanges in `split_redist_balance`, and the series of reductions in `max_var_quick_select`.

### 6.6.1 Pipelining Mechanism

We introduce a pipelining mechanism [CCGV19] to stagger the communication within the phases in iterations of Pseudocode 7. This pipelining mechanism enables communication/computation overlap for multiple (enough) iterations. The intention is not to



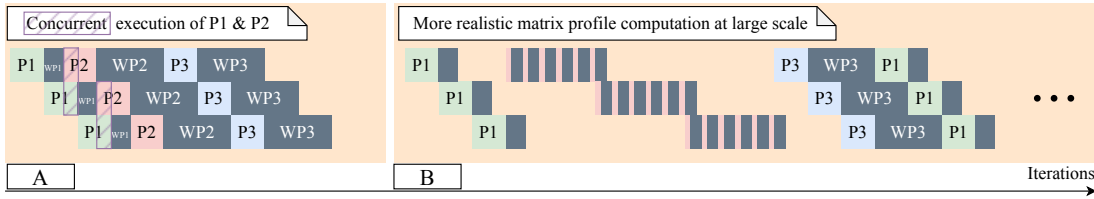
**Figure 6.4:** Pipelining the iteration phases (P1, P2, and P3) in Pseudocode 7. Blocking *wait* operations for phases are annotated with WP1, WP2, and WP3.

introduce more compute parallelism but rather optimize the existing parallelization by hiding communication latencies.

Such pipelining mechanisms are well-known techniques for hiding communication latencies of phases in parallel SPMD applications. Specifically, the shallow pipeline with a length of one, i.e., staggering a computation and a communication phase is a straightforward optimization in parallel programs and is typically enabled through non-blocking communication calls. In the case of the tree-based matrix profile computation, lengthier (i.e., deep enough) pipelines are required to enable latency hiding on a larger setup: for the runs on large portions of the HPC systems, the network latency become the bottleneck (e.g., in a fat-tree network topology of the target system). In this case, staggering multiple phases and/or deeper pipelines is beneficial.

Figure 6.4 provides a simplified schematic representation for this pipelining scheme. We illustrate how the phases of Pseudocode 7 are pipelined, where the benefit of deeper pipelines is visible. While this pipelining mechanism applies to an SPMD program, for simplification, we are only sketching the pipeline for the phases spawned by a single execution process (i.e., a single MPI process). **We use the parameter  $l$  to represent the pipeline length (depth)**, as the main configuration setting for the pipelining mechanism, allowing for flexible pipelining of multiple ( $l$ ) iterations.  $l$  is set to three for the sketch in Figure 6.4. The value of  $l$  is set according to the experimental setup and tuned based on the overall scale of the problem and the performance behavior on the system. Each phase consists of a blocking *wait* call (annotated with WP in Figure 6.4) to ensure the completion of communications of a phase at the start of its successive phase.

Figure 6.5 provides more details about this pipelining mechanism: in fact, the pipelining mechanism can have a potential drawback in the performance of the phases that are, in the end, co-scheduled for *concurrent* execution on the same computing resources. An example of such a scenario is illustrated in Figure 6.5 left [A]. We are explicitly preventing this scenario to avoid the concurrent execution of multiple phases on the same resource. Moreover, in our evaluations, we observe that the *all-to-all* communications (i.e., *wait* operations corresponding to MPI\_Ialltoall) cause major communication bot-



**Figure 6.5:** **A**: a scenario with potential performance drawbacks due to concurrent execution of P1 and P2 on the same resource. **B**: more realistic sketch of the pipeline and latencies.

tlenecks at large scale. Therefore, at this stage, we end up with a pipeline that looks like the sketch in Figure 6.5 right **B**. Overall, the pipelining mechanism staggers the *all-to-all* communication latencies along with the computation of other phases. This allows the overlapping of computation and the aforementioned *all-to-all* communication latencies and prevents the run time from being dominated by the *all-to-all* latencies in large jobs.

Pseudocode 8 describes the pipelining mechanism in more detail. However, for conciseness, we only present the pipelining mechanism for Phase 3, i.e., `merge_iteration_results_all2all`. Specifically, we discuss the non-blocking *all-to-all* communications and their corresponding wait operation (the same can be applied to other phases as well). In Line 1, we initialize the wait handles as well as separate buffers to enable concurrent execution of  $l$  iterations. In Line 2, we start the main iteration loop in which the iterator  $i$  is incremented by the pipeLine length  $l$ . A nested loop triggers the concurrent execution of  $l$  iterations (Line 7). In particular, in the  $j$  loop in Line 7, after calling to `merge_iteration_results`, the asynchronous communications are spawned ( $l$  times). We then call the corresponding wait operations `wait_all_pipes` (`wait_handles_p3`) in the next iteration of the  $i$  loop at the latest point possible.

---

**Pseudocode 8** Pipeline mechanism to stagger the *all-to-all* communications

---

**Configuration** ( $l$ ) represents the pipeline depth.

```

1: wait_handles_p3 = Initialize_pipeline ()
2: for i ← 0 to T+1 with step l do in pipes
3:   ... ▷End of P1
4:   wait_all_pipes (syn_handles_p3) ▷Wait WP3
5:   ... ▷End of P2
6:   ... ▷(Details are removed) P3
7:   for j ← 0 to l do concurrently
8:     merge_iteration_results()
9:     launch_comms_async (wait_handles_p3[j]) ▷Trigger Comm P3
10:  end for
11: end for

```

---

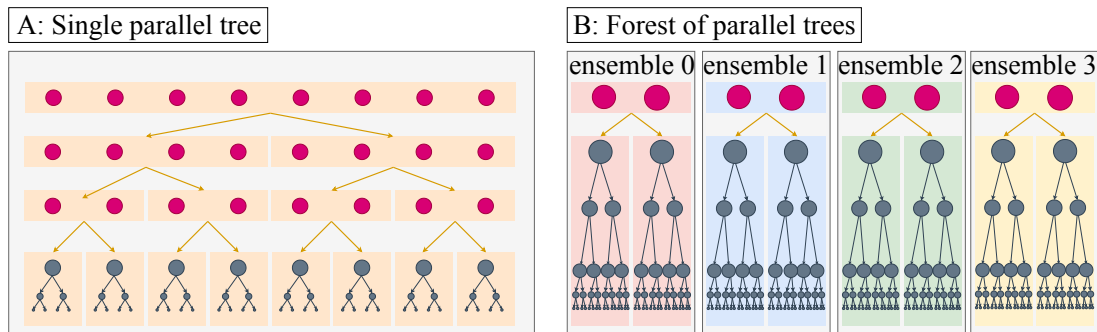


As we target HPC systems, the pipelining mechanism described in Pseudocode 8 is realized using MPI, leveraging non-blocking *all-to-all* communication, i.e., in Phase 3, MPI\_Ialltoall and MPI\_Waitall operations are used.

### 6.6.2 Forest of Trees on Ensembles of Resources

While pipelining is promising for improving the performance of Phases 2 and Phase 3, it is not suitable for the optimization of the communication in Phase 2, namely staggering the communications in the series of **reductions** and pair-wise data exchanges (**redistribute** and **balance**) during the tree construction and embedding phase (2): the main reason is that pipelining the tree construction loop in Line 6 in Pseudocode 7, requires another level of nested pipelining due to the dependency of the iterations of this loop. However, creating another pipeline level would be infeasible in practice due to excessive memory consumption overheads of nested pipes, as each level of the pipeline requires the allocation of extra buffers.

We take a different approach to address these overheads. The key idea here is to utilize the parallelism in the outermost loop at Line 2 of Pseudocode 7. Therefore, instead of using all the resources for constructing a *single* tree (Figure 6.6 left), we construct a forest of *multiple* trees of a certain size (four in Figure 6.6 right) which works in parallel on exclusive ensemble of resources (nodes/processes). Each tree is built on an (exclusive) subset of resources (nodes/processes) and runs in parallel to other trees. For example, in Figure 6.6 right, the illustrated work corresponds to four iterations, each assigned to one of the four ensembles, which then run independently in parallel. The idea resembles the class of parallel in-time [HSM20] methods, where the time dimension in solvers (main iteration loop) is used as an extra knob for parallelization. However, since there is no explicit time discretization applied here and the iterations are randomized, there is no dependency between iterations, and therefore, the parallelization is even simpler. Therefore, when employing the forest mechanism, the math and underlying structure of the computation approach presented in Pseudocode 7 stays the same.



**Figure 6.6:** Single parallel tree vs. forest of trees. Pink circles represent the resources, e.g., processes. Rectangular enclosures with unique colors represent the communication contexts (i.e., MPI communicators) at different levels of a parallel tree. Each parallel tree is represented by a hierarchy of boxes with a unique color.



---

**Pseudocode 9** Forest of trees on ensembles of resources

---

**Configuration** ( $e$ ) represents the number of ensembles.

---

```

1: C_ensemble, C_trans = create_ensemble_comm_context (e)
2: Part = partition_iterations (e)
3: for itere = Part.start to Part.end do on ensembles
4:   ... ▷Run P1 on C_ensemble
5:   ... ▷Run P2 on C_ensemble
6:   Ie, Pe = P3 (C_ensemble) ▷Run P3 on C_ensemble
7: end for
8: I, P = merge_partial_results ({Ie, Pe}, C_trans)

```

---

Since nodes/processes in each ensemble work on a specific logical tree, the communications remain local in each ensemble. Therefore, the nodes/processes sitting in different ensembles do not communicate across (i.e., communication only happens among the nodes/processes in the same ensemble). For this reason, this scheme reduces the overall communication overheads.

Also, this scheme allows coarsening the parallelism in the inner computations and communications (Lines 4-13 in Pseudocode 7), including the partitioning involved in the *KD-tree* (Phase 2 of Pseudocode 7). Specifically, it allows the *reductions* and *pair-wise data exchanges* in *redistribute* and *balance* as well as the *collectives* in other phases to run on coarse-grained parallelism in smaller communication contexts (i.e., communicators). This can also be observed in the example in Figure 9: the largest communication context (at the top of the tree) in Figure 9 left includes eight processes/nodes, while the forest approach uses only two processes/nodes, at the top level of the tree. The main caveat for this approach is the high memory consumption, which scales with the number of ensembles.

Pseudocode 9 provides the details of the forest of trees approach. We split the available resources into smaller sets, i.e., *ensembles* (Line 1 in Pseudocode 9). Ensembles have the same amount of resources (processes/nodes). Assuming fairly uniform iterations, we statically partition the iterations of the loop in Line 2 of Pseudocode 7 and assign them to these ensembles (Lines 2 and 3 in Pseudocode 9). We then run the iteration phases on these ensembles (Lines 3-7). We perform a final merging step across the ensembles (Line 8) to aggregate partial matrix profiles in ensembles. This is optionally done during the iterations to estimate the overall accuracy among all ensembles.

We implement this approach in *MPI* by splitting *MPI*'s default communicator using `MPI_Comm_split`<sup>1</sup>, where the resulting communicators represent the corresponding en-

---

<sup>1</sup>`MPI_Comm_split` partitions a group of *MPI* processes associated with a communicator into disjoint subgroups and creates a new sub-communicator for the subgroups.

sembles and the communication context for each tree. The aggregations in the final merging step (Lines 8) are realized using `MPI_Allreduce`.

## 6.7 Scaling Behavior and Limitations of Tree Approach

To better understand the performance characteristics of the tree-based method, we reuse the complexity analysis of Xiao et al. [XB16] and adapt it according to the pipelining and forest mechanisms in Appendix A2.

According to this analysis, Phase 1 is the least problematic part; it only includes the parallel transformations of data as well as a broadcast. The pipelining mechanism has the capability to stagger and eliminate the overheads of the reduction (the term highlighted in red) in this phase, and the forest mechanism can reduce the size of the communication context of the reduction operation. In Phase 2, only the forest mechanism affects the size of the communication contexts and slows down the growth of the overheads by a **factor  $e$ , which is defined as the size of the forest**. In Phase 3, the pipelining mechanism staggers the overheads of *all-to-all* operations into other phases. Also, similar to the other phases, the forest mechanism has a similar effect on reducing the communication context for the *all-to-all* operations (factor  $e$ ).

Also, note the time to run the iterations of the tree-based approach, with optimizations plugged in, is similar to its parent randomized `KD-tree` method, i.e., regardless of the parameters chosen for the pipelining and forest mechanisms, the costs of the iterations of the tree-based method is sub-quadratic. This results in overall sub-quadratic costs in the number of records of input time series, unlike `SCRIMP++`, which overall scales quadratically with problem size. However, the randomized tree-based approach requires multiple iterations, i.e., multiple rounds of creation of trees and searching in them. Therefore, there is an additional multiplier in the overall time complexity and overheads of the tree-based approach. With some assumptions on the distributions of the `z-normalized` subsequences, similar to those made in [RS19], we can assume that running the tree-based approach in a certain number of iterations can reach a certain accuracy level and eventually converge to the exact solution. Analyzing these conditions falls out of the scope of this chapter, but what we want to highlight here is that the overall performance of the tree-based method relies on the content of the time series dataset, as well as the required accuracy level. If the exact solution is needed, the tree-based approach would not be the right scheme.

Also, note that, unlike `SCRIMP++`, the tree-based approach cannot benefit from the affinity of neighboring subsequences in the evaluation of distances and does not allow the use of the streaming dot-product formulation [ZKS<sup>+</sup>19]. Therefore, there is a linear growth in the run time of the tree-based approach with windows size  $m$ . This suggests that the tree approach is more suitable for large time series (large  $N$ ) and mining of fine-grained patterns (small  $m$ ). This, however, is a frequent problem setup in analysis use cases where the subsequence length is much smaller than the time series length  $m \ll N$ . We summarize a list of such use cases from literature in Table 7.2, which we will discuss later.

## 6.8 Evaluation

We evaluate the suitability of the tree-based approach and assess its accuracy when it is used to compute the matrix profile index in real-world datasets. We look at the scaling overheads of tree approaches and show how the optimization schemes presented in this chapter improve scalability.

### 6.8.1 Experimental Setup

We briefly describe our prototype implementation for the tree-based matrix profile approach introduced in the chapter and further elaborate on the experimental setup used for the evaluations presented.

We implement the tree-based matrix profile approach in `C++` using pure `MPI` for parallelization<sup>2</sup>. Our implementation targets `CPU`-based `HPC` systems, and it relies on vectorized sorting and searching kernels as well as optimized `DGEMM` kernels in the Intel `MKL` for distance computation. In all experiments, we use the Intel `OneAPI` package v21.2.0, including the Intel `C++` compiler v21.2.0, with the highest optimization level (`-Ofast`) for compilation, and the Intel `MPI` v21.2.0. Our targeted system for the experiments of this chapter is again the `SuperMUC-NG` system at the `LRZ`.

In all the evaluations presented in this chapter, we quantify of the quality of the results by comparing the matrix profile index against the exact computation, i.e., brute force computation. For this comparison, we use recall ( $\mathcal{R}$ ) on the matrix profile index, and reuse the same definition presented in Section 5.5. Note that other metrics are also used in the literature, and some might be subjective to particular practical use cases, but here, we rely on a generic accuracy metric. An investigation of such alternative metrics is beyond the scope of the present work.

In all the cases, we report the performance or accuracy metrics for the average of 5 repeated runs, and do not present statistical errors where they are insignificant.

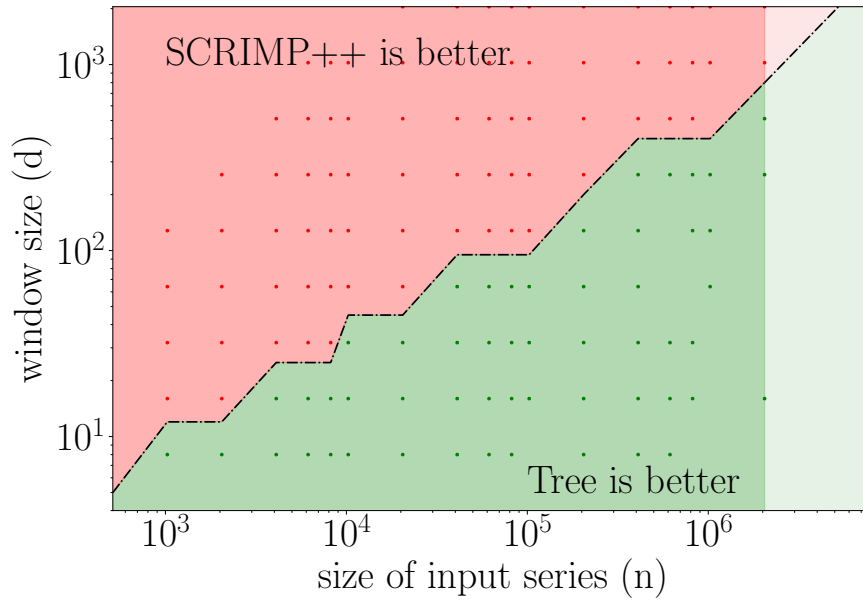
### 6.8.2 Baseline Comparison and Benefit Margin for Tree Approach

We use the `C++` implementation of `SCRIMP++` [ZYZ<sup>+</sup>18] as the baseline to compare the tree-based method. Due to the limitations of other approximate approaches, discussed in Section 6.3, we are limiting these comparisons to `SCRIMP++`. In particular, in all these implementations, the parallelization support is limited, and often, the analysis of large datasets is at least an order of magnitude slower in comparison to the parallel tree-based method [LWM<sup>+</sup>22]. We are also excluding the downsampling of the input time series as an approximation method in our evaluation, as it is out of the scope of matrix profile computation methodologies. Finally, we are not comparing the performance of the tree-based method to any exact method but `SCRIMP++`, as the performance characteristics of all these methods are similar to `SCRIMP++`.

Looking back at the discussion in Section 6.4 and, the performance advantage of the tree-based method directly depends on the size of the two input time series  $n$ , and the

---

<sup>2</sup>See Appendix A4 for pointers to the code.



**Figure 6.7:** Illustration for the benefit margin for the tree approach compared to [SCRIMP++](#). Red dots (and region) represent the areas where [SCRIMP++](#) has better performance, while the green dots (and region) represent the areas where the tree-based method is superior.

windows  $m$ . Therefore, we use random walk datasets with different combinations of  $n$  and  $m$  parameters to compare the two methods. For a fair comparison, we fix the accuracy level and compare the time to get to at least  $\mathcal{R} = 99.99\%$  accuracy for both methods. Finally, for fairness, we compare the single-thread performance of the two methods.

Figure 6.7 illustrates the results of this experiment, providing a visual representation to quantify the benefits of the tree approach. Each dot on this graph represents one combination of the input parameters ( $n$  and  $m$ ), and it is color-coded as green if the combination results in faster computation using the tree approach and red if [SCRIMP++](#) is faster. Therefore, we clearly see the region where the tree approach is more beneficial, and the dashed line shows its limits.

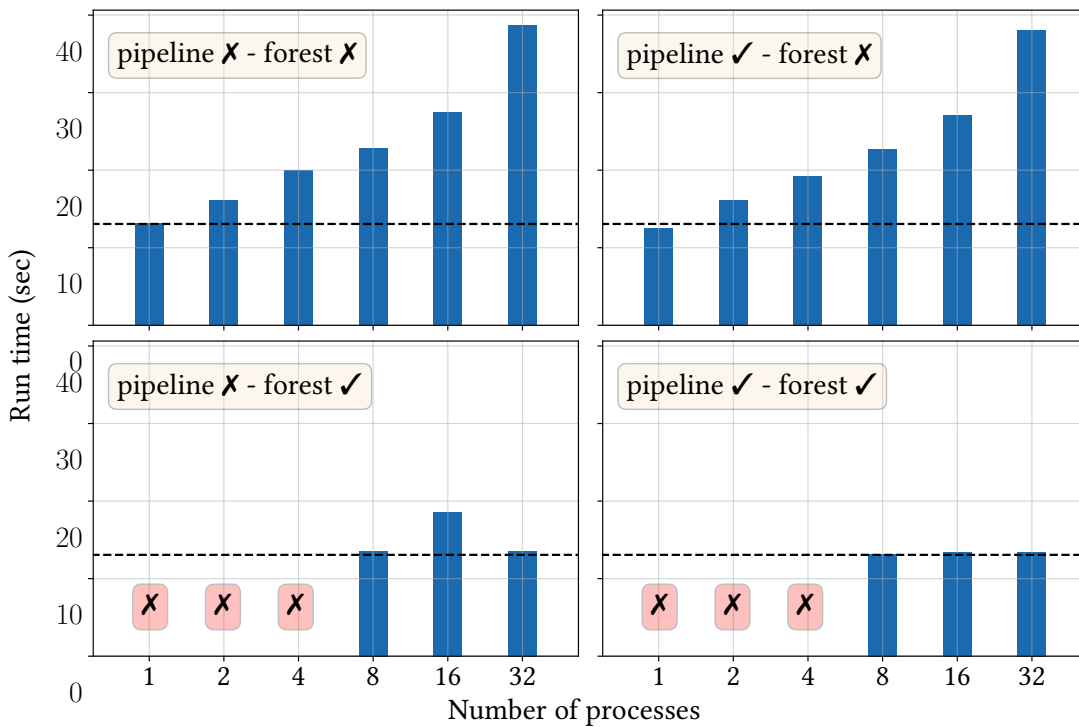
We observe that as the size of the input datasets increases relative to the window size (towards the right side in the graph), the tree-based approach becomes more beneficial as the green region becomes wider. This is also in accordance with the discussions in Section 6.4.

While the tree method cannot benefit from the streaming dot product, in practice, it might outperform other methods that employ this formulation, depending on the problem configurations (i.e., the configuration lies on the region of benefit).

Note that to keep this region of benefit relevant for large input time series, the parallel efficiency of the tree-based method should be high. One of the main contributions of our work is to address the high parallel efficiency of the tree-based method through the approaches discussed in Section 6.6 and to maintain the region of benefit consistent for large time series.

### 6.8.3 Performance of the Tree Approach

We start by running a set of weak scaling experiments on a single node of the SuperMUC-NG system. Figure 6.8 summarizes these experiments. We present four different cases where the pipelining and forest optimizations are either enabled or disabled. On the top left graph, we show the run time and overheads of the tree-based method when scaling on a single node. We observe large overheads ( $\approx 100\%$ ) even on such a single node. Digging further into the overheads, we notice that the sources of them are mainly in Phase 2 and Phase 3 of Pseudocode 7. Specifically in all the cases `max_var_quick_select` and `merge_iteration_results_all2all` comprise at least 60% of the run time. However, `merge_iteration_results_all2all` is only responsible for a small portion of it, and as we will show, this overhead only appears in larger setups.



**Figure 6.8:** Single-node performance of the tree approach, with ( $\checkmark$ ) and without ( $\times$ ) pipelining or forest mechanism ( $n = 6K$  per core,  $m = 512$ , random walk dataset). Configurations annotated with  $\times$  mark are *algorithmically* not supported.

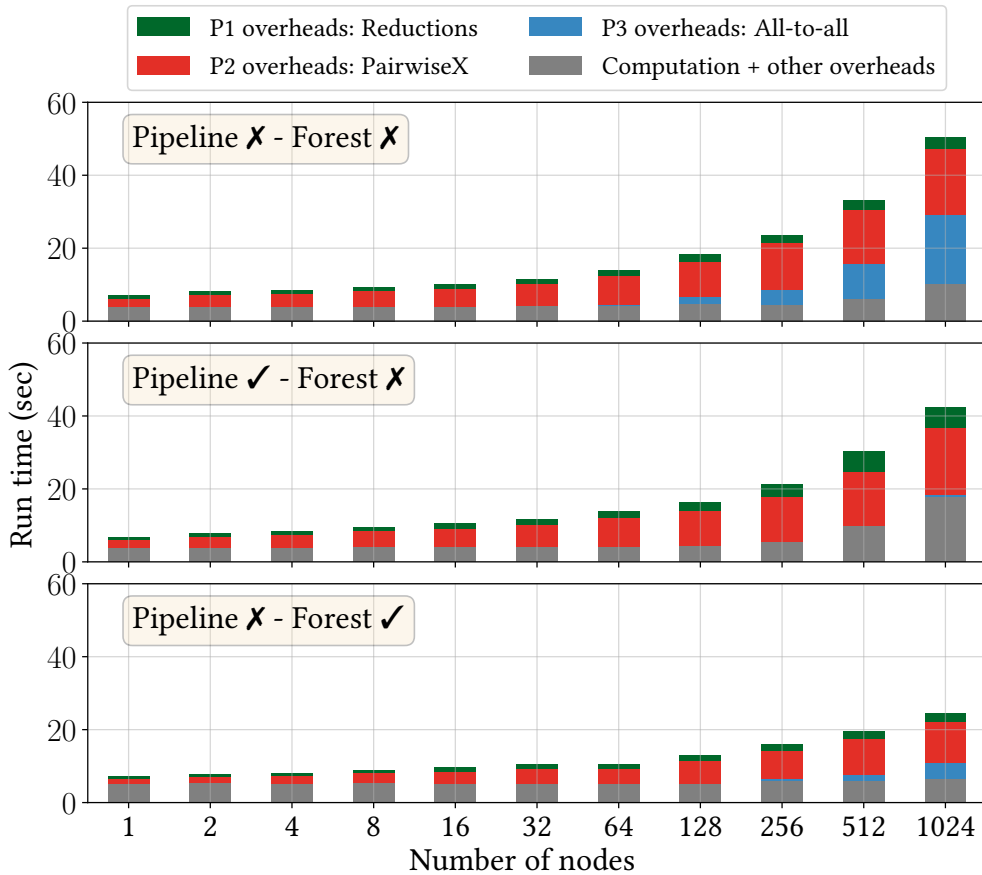
## 6 Other Tackling Point; Algorithmic Approach for High-Performance Similarity Mining

We repeat the experiment and enable the pipelining mechanism with a depth of four (top right). We observe almost negligible improvements. Therefore, the pipelining mechanism does not seem to bring any advantage to single-node runs.

However, we observe on the graph at the bottom left that when enabling the forest mechanism and setting the size of the forest (the number of ensembles) to eight overheads significantly reduce. Finally, when we combine the two mechanisms (bottom right), we almost completely overcome the overhead and reach  $\approx 100\%$  parallel efficiency on a single node.

Overall, the proposed optimization schemes are able to eliminate the scaling overheads on a single node of the [SuperMUC-NG](#) system.

We continue the performance evaluation of the tree approach by going beyond a single node and looking into the system-wide scaling overheads.



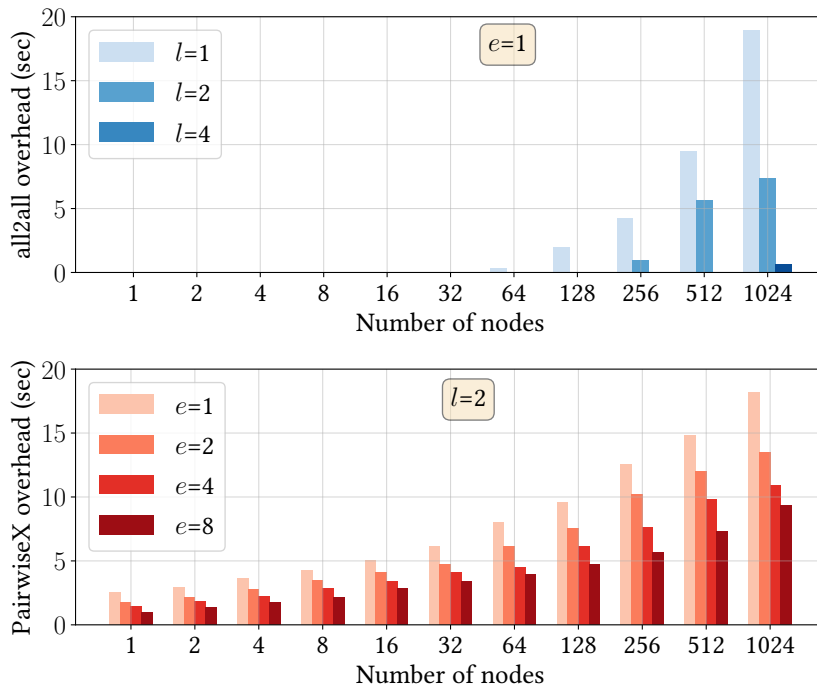
**Figure 6.9:** Breakdown for the time spent in various phases, with (✓) and without (✗) pipelining or forest mechanisms, in weak scaling on [SuperMUC-NG](#). ( $n = 4K$  per core,  $m = 256$ , and random walk dataset).

We run a set of weak scaling experiments on the [SuperMUC-NG](#) system with a configuration starting from a single node up to 1024 nodes. On the Top graph in Figure 6.9, we present the breakdown for the time spent in different phases of Pseudocode 7 for computing the matrix profile. As we scale the problem and resources, the overhead in `max_var_quick_select`, `merge_iteration_results_all2all`, and `split_redist_balance` scales drastically; When we use 1,024 nodes, the overheads comprise more than 75% of the run time, where the largest overhead is associated with the *all-to-all* operations in `merge_iteration_results_all2all`.

We repeat the same scaling experiments and switch the two mechanisms on, one at a time. Once the pipelining mechanism is switched on (middle graph in Figure 6.9), the overhead of *all to all* operations is removed. However, this optimization alone only has around 9% performance improvements. On the other hand, once the forest mechanism is enabled (bottom graph in Figure 6.9), the overheads of all the phases are shrunk much more, i.e., by  $\approx 55\%$ .

Overall, the proposed optimization schemes are able to reduce the scaling overheads in a system-wide weak scaling on the [SuperMUC-NG](#) system.

To better understand the effects of the two optimization mechanisms and their interplay, we conduct a series of weak scaling experiments on the [SuperMUC-NG](#) system



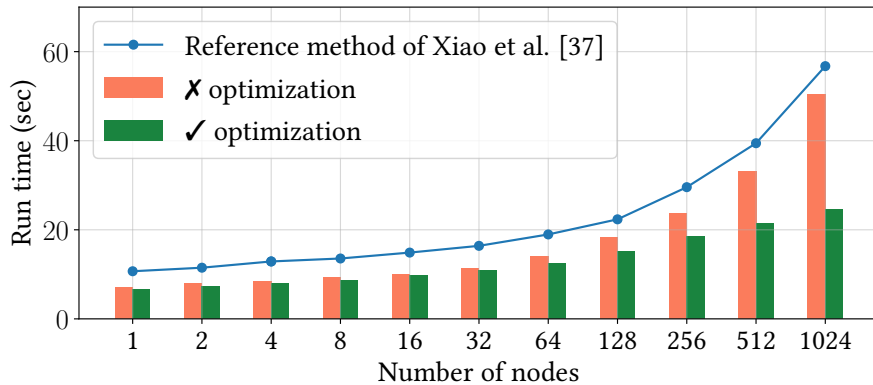
**Figure 6.10:** Effect of pipelining and forest mechanisms on scaling overheads ( $n = 65K$  per node,  $m = 256$ , 32 iterations, and random walk dataset).

with various pipeline depth ( $l$ ) and forest size ( $e$ ) configurations. Figure 6.10 summarizes the results of these experiments: in Figure 6.10 Top, we revisit the overhead of the *all-to-all* operations in `split_redist_balance`, which only appears in the setups larger than 64 nodes. We observe that in all the cases, enabling the pipelining mechanism is highly beneficial, and increasing the depth of the pipeline can help to almost completely eliminate the *all to all* overheads. In Figure 6.10 Bottom, we focus on the overhead of `split_redist_balance`, which was the main target for enabling the forest mechanism. We observe that increasing the size of the forest in all the experiments improves the performance. Also, similar to the results shown in Figure 6.9, we also observe positive effects on other overheads (not present in Figure 6.9).

Also, note that in Figure 6.10 Bottom, the pipeline depth  $l$  is set to 2. Our further experiments suggest enabling them simultaneously and tuning  $e$  and  $l$  parameters is beneficial.

The two optimization mechanisms have a complementary effect on performance improvements.

Finally, we conduct a weak scaling experiment on the SuperMUC-NG system to illustrate the overall scaling performance of the tree-based matrix profile approach. In this experiment, we use two cases; in the first case, we disable all the optimization mechanisms and in the second case we enable the pipelining mechanism with depth  $l = 4$  as well as a forest of size  $e = 4$ . Figure 6.11, illustrates the results of this experiment: again, we observe that the tree-based approach results in scaling bottlenecks if no optimization is used. In this case, starting from one node and increasing the number of MPI processes by three orders of magnitude on SuperMUC-NG, i.e., going beyond a single island, results in 14% parallel efficiency, which is also in accordance with the experiments of Xiao et al. [XB16].



**Figure 6.11:** Weak scaling results with (✓) and without (X) optimizations ( $n = 65K$  per node,  $m = 256$ , 32 iterations, random walk).



Overall, applying the two pipelining and forest optimization mechanisms improves efficiency to 31%, which is a 2x improvement on the SuperMUC-NG system.

However, this efficiency shrinks the region of benefit (see Figure 6.7), however overall, the tree method stays beneficial for larger time series, as discussed in Section 6.8.2. Finally, we trace the remaining scaling overheads to the series of pairwise exchanges in `split_redist_balance`. Although the forest mechanism improves the overheads of data shuffling and load balancing in Phase 2 of Pseudocode 7, it still does not entirely eliminate the latencies of the blocking calls in this phase.

In the end, we showcase the capabilities of the tree-based approach by running a matrix profile computation on a billion-record random walk dataset ( $n = 1$  billion) with a window size  $m = 128$ , using 48K cores of the SuperMUC-NG to 99% accuracy in 19 minutes, which is 3-100 times faster than existing approaches. This run reaches  $\approx 338$  teraflop/s maximum kernel performance. This run also demonstrates the significant improvement of the tree approach over existing approaches for the computation of the matrix profile. More details on this experiment are presented in Appendix A3.

## 6.9 Summary

In this chapter, we presented an approximate parallel iterative tree-based approach for the computation of matrix profiles, targeting CPU-based HPC systems. We analyzed the scaling bottlenecks state-of-the-art tree-based approach for matrix profile computation in our work. We provided pipelining and forest mechanisms to improve the scalability of the tree-based approach. Our optimizations improved parallel efficiency on 32K cores by a factor of 2. The optimization mechanisms presented can be generalized and applied to similar problems (e.g., nearest neighbor computations or randomized iterative algorithms).

With the presented work, it is feasible to compute the matrix profile for large time series, which previously required days to compute, in just a few minutes. We showed the capabilities of this tree approach by presenting a billion-scale experiment with up to 99% accuracy in 19 minutes, which is 3-100 times faster than existing approaches. This provides a competitive computation time compared to the alternative, in particular, exact approaches.

## 7 Practical Use Cases and Real-World Examples

The examples described in this chapter were previously published in IPDPS'22 [JRY<sup>+</sup>22] and ISC'23 [RKS<sup>+</sup>23], where the author of this thesis made a major contribution to the contents, including the development of the prototype codes, evaluations.

In this chapter, we delve into several illustrative use cases and examples to demonstrate how the concepts and experimental methodologies introduced throughout this thesis translate into practical applications. Building upon the foundation laid out in Chapter 2, we revisit the representative examples presented there as case studies to showcase the versatility and power of the methodologies proposed in this thesis.

These practical scenarios include the analysis of patterns within the operational data from heavy-duty gas turbines, and analysis of application patterns in HPC operation data, as well as the accuracy of the tree method introduced in Chapter 6 on a representative number of real-world datasets and analysis configurations. These examples offer a practical perspective on the real-world impact and significance of our research. Moreover, these examples help to explore the intricacies of applying matrix profile computation techniques introduced in this thesis in different contexts, shedding light on the flexibility and robustness of the proposed methods.

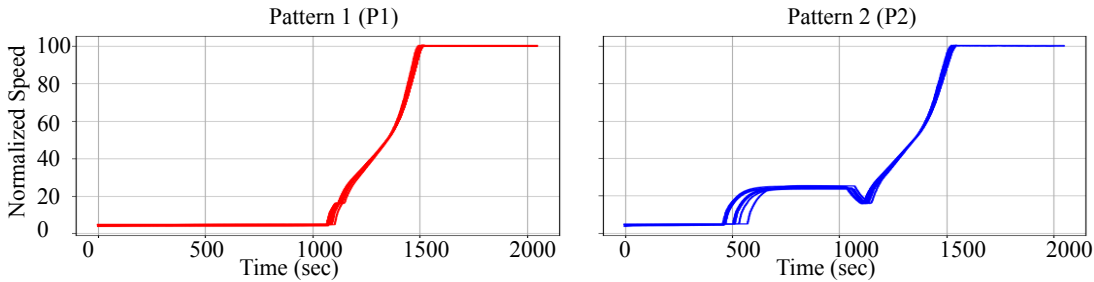
We start with the first use case on the analysis of patterns in the operational data of heavy-duty gas turbines, which helps to better understand the benefits of (MP)<sup>N</sup>-GPU approach in practice.

### 7.1 Use Case 1: Patterns in Operational Data of Heavy-Duty Gas Turbines

In the first use case, we look at the operational data from the surveillance systems for large-scale heavy-duty gas turbines and investigate the accuracy of pattern detection based on the matrix profile approach when (MP)<sup>N</sup>-GPU is used for computation.

With the increasing growth of renewable energy sources in power grids, heavy-duty gas turbines are often operated as a backup source for power generation. Consequently, they are exposed to more dynamic operation settings and, therefore, potentially to frequent *startups* and *shutdowns* cycles. This dynamic setting imposes extra operational pressure on the infrastructure, which it was not necessarily designed and optimized for. Therefore,

## 7.1 Use Case 1: Patterns in Operational Data of Heavy-Duty Gas Turbines



**Figure 7.1:** Startup patterns in heavy-duty turbine datasets. We apply min-max normalization to avoid overflow in reduced- and mixed-precision computation.

inspecting these patterns within the operational data of turbines is interesting for domain experts, as it helps to identify similar *startup* and *shutdown* events appearing as unique patterns. Furthermore, it helps them to look into the corresponding high-frequency dynamic monitoring data that is collected from the combustion surveillance sensors for a better understanding of the combustion process under extra stress. This helps to protect the infrastructure with predictive control schemes and improve future designs.

In order to better detect and later predict such events, we investigate turbine speed data collected directly from the machine in a real-world power plant with all its real-world limitations. We represent the data as a time series and especially focus on the detection of startup events. This use case is a special case where the dimensionality of input time series,  $d$  is one, but using reduced- and mixed-precision in  $(MP)^N$ -GPU is still a key to improving the performance. We look at recall rate ( $\mathcal{R}_{practical}^r$ ) as an accuracy metric when applying the matrix profile analysis approach to pattern detection.

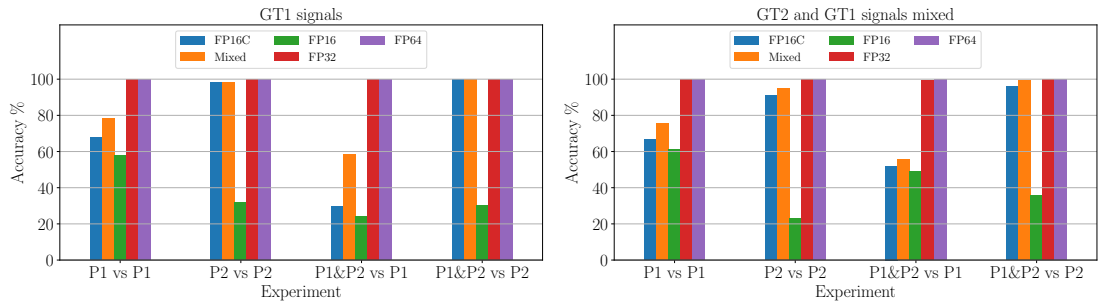
We exploit a dataset derived from the actual operation of two instances of gas turbines, here labeled **GT1** and **GT2**, installed and operated by a large municipal power provider. Figure 7.1 illustrates examples of the startup events (annotated with  $P1$  and  $P2$ ) in our dataset, each corresponding to a specific operation initiation mode. We use a matrix profile on the combinations (pairs) of small time series that include such startup patterns. Specifically, we prepare 65 single time series ( $n = 2^{16}$ ,  $m = 2^{11}$  and  $d = 1$ ) that include either  $P1$  or  $P2$ , respectively, and five single time series that include both.

We run a matrix profile analysis between the combinations of these input series structured in four classes (i.e.,  $P1$ - $P1$  and  $P2$ - $P2$ , both- $P1$ , and both- $P2$ , as highlighted in Table 7.1). The numbers of input time series pairs in each class are listed in Table 7.1.

	<b>P1-P1</b>	<b>P2-P2</b>	<b>both-P1</b>	<b>both-P2</b>
<b>GT1 or GT2</b>	4160	4160	325	325
<b>GT1 and GT2</b>	4225	4225	650	650

**Table 7.1:** Categories of time series pairs, and the numbers of input time series pairs in each category in the gas turbine use case.

## 7 Practical Use Cases and Real-World Examples



**Figure 7.2:** Recall rate ( $\mathcal{R}_{practical}^r$ ) for (MP)<sup>N</sup>-GPU with various precision modes for detecting startup events in the gas turbine dataset.

This experimental setup is sufficiently large and diverse to analyze the startup pattern detection accuracy within one or across the two turbine instances.

Following the above experiment settings, we compute the success rate of pattern detection according to the definitions provided in Section 5.5 for computing the *practical* accuracy.

For the P1-P1 case, we run  $65 \times 64 = 4,160$  individual matrix profile computations (corresponding to the first row in Table 7.1) and identify the success rate (*practical* accuracy) in detecting P1, e.g., for the data from either GT1 or GT2. In the same way, we analyze P2-P2 on either the data from GT1 or GT2 data to identify the *practical* accuracy ( $\mathcal{R}_{practical}$ ) for detecting P2. The experiments labeled with both-P1 and both-P2 analyze the five prepared time series data that include both the P1 and P2 patterns against those that exclusively include either P1 or P2, respectively.

We observe that both FP64 and FP32 exhibit  $\mathcal{R}_{practical} = 100\%$  accuracy in all the cases. Mixed and FP16C provide higher accuracy compared to FP16. Overall, this use case demonstrates a satisfying pattern detection accuracy for FP64, FP32, Mixed, and FP16C, showcasing the capabilities of (MP)<sup>N</sup>-GPU.

### 7.2 Use Case 2: Application Classification on HPC-ODA

Next, we explore extending the applicability of matrix profile analysis with reduced- and mixed-precision modes to another interesting practical use case. In Section 5.6.2, we looked at the matrix profile analysis on the HPC-ODA dataset. Please also refer to Chapter 2, where we present a couple of more illustrative examples to introduce matrix profile concepts). We look at the HPC-ODA dataset, which is introduced in Chapter 2, and use the application classification example. We leverage the (MP)<sup>N</sup>-GPU approach in computing the matrix profile on the way to build a nearest neighbor classifier.

Various telemetries and application performance data are continuously collected in HPC centers. Analyzing the collected data can provide valuable insights into the operation of the supercomputers, as discussed in detail in Chapter 2. Such analyses are highly relevant for HPC center operators since they provide insights to further optimize the operation of the HPC center based on the characteristics of applications that frequently

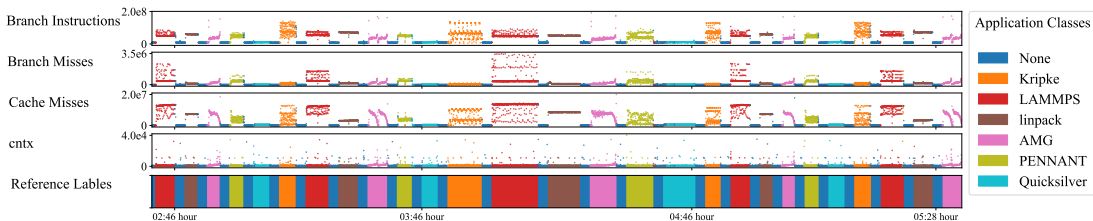
## 7.2 Use Case 2: Application Classification on HPC-ODA

run, e.g., via improved scheduling or preventing unwanted applications, just to name a few examples. One such approach to classify applications relies on gathered performance metrics, such as resource hardware counters or utilization patterns, collected during the application runs. Such data is often stored as large multi-dimensional time series (see Figure 5.9).

Specifically, to test the use of reduced- and **mixed-precision** computation modes and employing GPUs in (MP)<sup>N</sup>-GPU, we construct a nearest neighbor classification model on top of HPC-ODA data to enable the classification of HPC applications for the CORAL benchmarks [VdsB<sup>+</sup>18]. As discussed in Section 5.6.2, we use the *Application Classification* segment of a public HPC dataset [Net20]. This dataset includes labeled performance data collected while running different CORAL benchmarks (HPL, AMG, etc.) on 16 compute nodes for one day with one Hz sampling rate. We select 16 distinct sensors (performance metrics) on different nodes and split the resulting multi-dimensional matrix profile dataset along time into two portions, a reference set and a query set, each of which includes continuous operational data for half a day. More details of how the classification model works, as well as the details of its accuracy and run time.

Figure 7.3 shows an illustrative visualization of the HPC-ODA dataset, including a timeline for various sensor data collected. This visualization graphically shows a sample of the query dataset, and we highlight the application classes with distinct colors in the timeline. This visualization helps us to better grasp the high accuracy of the matrix profile computed with (MP)<sup>N</sup>-GPU in FP16C modes in practice. We are only presenting a subset of data used in this use case, both in the sense of the sensors involved and the time span. The sensor data (in the top four rows) are color-coded with the classifier’s predicted labels, and we use a color bar (lowest row) with consistent colors to annotate the ground-truth labels in the HPC-ODA dataset. Although the computation is performed in FP16C mode, the *F1* score of more than 99.4%. This represents a high-quality classification, which is also clearly visible in the presented timeline in Figure 7.3.

This use case proves that reduced- and **mixed-precision** schemes, such as FP16C mode, can achieve accurate results in practice while offering performance improvements as demonstrated in Chapter 5.



**Figure 7.3:** A timeline of HPC-ODA data color-coded with the classes determined from the nearest neighbor classifier.










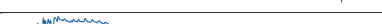

### 7.3 Real World Example for the Tree Approach: Benefits of on Real-World Datasets

Next, we take a look at the benefits of the tree approach compared to existing approaches when applied in real-world datasets and settings derived from real-world applications and scenarios. We extract these datasets and their corresponding configurations (in the sense of the size of datasets and subsequence lengths used in matrix profile analysis) from various sources in the literature and compare the performance of the tree approach using the implementation we presented in the previous chapter.

The datasets used in this example are listed in Table 7.2. We selected these diverse datasets carefully to cover a wide range of use cases and configurations (dataset size and subsequence lengths used in the analyses) to better illustrate the benefits of the tree approach presented in Chapter 5 based on the analysis provided in Section 6.8.2. Table 7.2 also illustrates a short snapshot of these datasets.

While various window sizes can be configured for matrix profile computation, we mainly use the configurations listed in Table 7.3, which reflects analysis scenarios commonly used in the literature. We present examples of such scenarios from literature in Table 7.3, together with the recommended subsequence length in the mentioned sources.

We compare the sequential<sup>1</sup> performance of our tree approach against **SCRIMP++** and **PreSCRIMP**. We specifically look at the time to reach a certain accuracy level as the performance metric and set this level to  $\mathcal{R} = 99.9\%$  accuracy. Table 7.4 summarizes the results of this experiment. Measurements suggest that our tree-based approach is overall faster than **SCRIMP++**, i.e., in eight out of eleven cases in Table 7.4 (highlighted in green) the tree approach outperforms **SCRIMP++**. The remaining three cases are exceptions where **SCRIMP++** achieves better performance compared to our tree approach. Note that these cases do not lie on the region of benefit for the tree-based methods that we demonstrated before in Chapter 6 in Figure 6.7 (compare the parameters listed

Dataset	Description	Dataset Source	Sample plot
HPC-ODA	HPC monitoring	[Net20]	
Earthquake	Earthquake vibrations	[ZKS <sup>+</sup> 19]	
Penguin	Penguin behaviour	[ZINK17]	
ASTRO	Celestial objects	[LZPK18]	
PSML	Energy grid	[ZXT <sup>+</sup> 21, LWM <sup>+</sup> 22]	
GT	Gas Turbine	[KWFH <sup>+</sup> 19, KRR <sup>+</sup> 21]	
GAP	Power consumption	[LZPK18]	
EMG	Electromyography signal	[LZPK18]	
InsectEPG	Insect behaviours	[YZY <sup>+</sup> 18]	
ECG	Electrocardiogram signal	[Eam22]	
MGAB	Synthetic anomalies	[TKB20, LWM <sup>+</sup> 22]	

**Table 7.2:** List of datasets used for the experiments and their sources.

<sup>1</sup>Note that it is very hard to set up a fair experiment to compare parallel implementations given the fact that at the time of conducting this research, no parallel implementation of **PreSCRIMP** existed.

### 7.3 Real World Example for the Tree Approach: Benefits of on Real-World Datasets

Dataset	n (number of records)	m (sub-sequence length)	Similar scenario in literature
HPC-ODA	34K	64	[JRY+22] Figure 8
Earthquake	100.00K	200, 100	[ZKS+19] Section 5
Penguin	1.000M	28	[ZINK17] Figure 10
ASTRO	1.100M	512	[LZPK18] Section 6.2
PSML	1.500M	200	[LWM+22] Figure 9
GT	1.900M	64	[KWFH+19] Figure 4
GAP	1.950M	512	[LZPK18] Section 6.2
EMG	2.860M	512	[LZPK18] Section 6.2
InsectEPG	6.400M	50, 256	[GYD+] Section E
ECG	19.2M	421, 256, 94, 128	[Eam22],[RCM+12] Table 5 & [YZU+16] Table 7, [LWM+22] Figure 13, [MAA+21] Figure 16
MGAB	112.5M	40	[LWM+22] Figure 4

**Table 7.3:** Problem settings associated with each dataset used in our experiments.

in Table 7.3 to regions depicted in Figure 6.7). Therefore, the better performance of SCRIMP++ in these cases is expected. Furthermore, we observe that for none of the datasets, PreSCRIMP is able to achieve 99.9% accuracy. Another observation is that the tree-based approach is superior for the datasets with a larger number of records, which are shown in the lower rows of Table 7.4. This is again in accordance with the results presented in Figure 6.7, and proves the superiority of our tree-based approach in various problem settings. However, real-world cases exist, similar to the first rows of Table 7.3 or the experiments in [ZYZ+18], where our tree approach is not superior.

Overall, the presented results validate that when applied to real scenarios, the tree approach can achieve high accuracy in a reasonable time, and it is superior to existing approximate methods in many cases, in particular against SCRIMP++ and PreSCRIMP for most of the considered datasets.

Dataset	Time (s) To 99.9% Accuracy		
	PreSCRIMP	SCRIMP++	Tree
HPC-ODA	✗	8.96	16.88
Earthquake	✗	13.47	32.62
Penguin	✗	7070.79	245.15
ASTRO	✗	1156.18	2615.70
PSML	✗	5936.69	303.60
GT	✗	7875.80	3104.13
GAP	✗	3320.65	2482.38
EMG	✗	7648.17	2874.43
InsectEPG	✗	9474.30	1282.58
ECG	✗	6060.12	1240.75
MGAB	✗	10749.50	20.67

**Table 7.4:** Time to 99.9% accuracy among various methods (single-core execution).

## 7.4 Summary

In this chapter, we briefly discussed a number of illustrative use cases and examples. In particular, we demonstrated the benefits of our (MP)<sup>N</sup>-GPU and our tree approach, which were discussed in previous chapters, in practice. These presented schemes have been put to the test in real-world scenarios to demonstrate their practicality.

We revisited the use cases from large-scale industrial operational data analytics and demonstrated how the methods presented in this thesis can help conduct practical tasks such as pattern detection and classification in practice with sufficient accuracy. We also illustrated the benefits of the tree approach in comparison to the existing methods and demonstrated its superiority when applied to many datasets from real-world scenarios.

In summary, the schemes presented in this work offer a significant leap forward in the realm of state-of-the-art time series mining using the matrix profile.



## **Part III**

# **Discussions, Wrap-Up and Prospects**



## 8 Discussions and Lessons Learned

In this thesis, we explored the mining of time series datasets within the realm of **HPC** and supercomputing. We embarked on a journey that revolved around the matrix profile as a central component. The concepts discussed in this work, the prototypes, and experiments helped to unlock the potential of **HPC** in the computation of matrix profiles, encompassing key facets such as scalability, **GPU** utilization, reduced-precision arithmetics, and alternative iterative approximate solvers. In this chapter, we discuss the key insights achieved by investigating the presented approaches. Furthermore, we discuss how these individual aspects come together as a whole and how they can work together in practice. Finally, we enumerate constructive lessons learned from conducting research on these aspects.

### 8.1 Synthesis of Key Insights

This work looked at various aspects of matrix profile computation, encompassing distinct approaches to enhance its efficiency and applicability. These key aspects include considerations for the scalability of matrix profile computation on High-Performance Computing (**HPC**) systems, computation using Graphics Processing Units (**GPUs**), exploration of reduced- and **mixed-precision** schemes, and the leveraging of an approximate tree method. Also, we covered both the single and multi-dimensional time series mining cases in our exploration. We also carefully investigated the practicality and limitations associated with each of these approaches, drawing insights from real-world use cases and examples. Through systematic experimentation, we showcased the tangible effects of our methods and provided a nuanced understanding of their implications.

We characterized the performance behavior of matrix profile computations, in particular, on high-end **HPC** systems. Our findings illuminated a crucial aspect: the primary bottleneck in this computation of matrix profiles lies in memory-bound operations. This is mainly due to the streaming dot product formulation, which is present in most approaches for computing the matrix profiles. Additionally, in multi-dimensional cases, sorting operations also add extra memory-bound operations, and they have an even more severe impact compared to the operations of streaming dot product formulation.

Among the various elements we investigated, the role of **GPUs** emerged as particularly instrumental. **GPUs** equipped demonstrated a significant capability in accelerating matrix profile calculations, providing a substantial overall performance boost, as their high memory bandwidth (due to their High Bandwidth Memory) improves the performance bottlenecks stemming from memory-bound operations mentioned in the previous paragraph.

We also investigated the realm of reduced- and [mixed-precision](#) schemes for the computation of matrix profiles. While these schemes enable enhanced performance, they come at a trade-off—sacrificing the level of mathematical accuracy. Nevertheless, we discovered that in practical applications, the results can still remain acceptable despite mathematical inaccuracies, making exploration of reduced- and [mixed-precision](#) schemes a valuable avenue for deployment in practice, especially when speed is of the essence.

We also investigated alternative tree-based approaches, which present a viable pathway for matrix profile computation. However, adopting a matrix profile to this alternative meant accepting solutions that were approximate in nature. However, this is what the tree-based approach and reduced- and [mixed-precision](#) schemes have in common. Despite this, the tree-based method shows promise, especially when enhanced for large-scale High-Performance Computing (HPC) systems. Our studies suggest leveraging the pipelining mechanism (introduced in Chapter 6) helps to mask communication latencies effectively. Furthermore, employing the forest mechanism (also introduced in Chapter 6) to manage the granularity of parallelization and communication allows for approaching a larger number of nodes in HPC systems. Together, these mechanisms can significantly mitigate the scaling bottlenecks of tree approaches on large-scale HPC systems. These improvements make the tree-based approach not only efficient but also an attractive option for matrix profile computation in practice.

Across the spectrum of concepts we investigated, the common thread that emerged was scalability as a central tenet. Our analysis of these various methods, problems, and setups reaffirms the claims made in the existing literature: the matrix profile is indeed a highly scalable approach, and the concepts we introduced in this thesis leverage this inherent property and make it feasible to scale matrix profile computation on large-scale HPC systems. This allows for employing the power of HPC for matrix profile computation for large-scale datasets. This also verifies the significance of scaling compute resources for matrix profile computation exploiting through HPC, where the main avenue of scalability hinges on the horizontal scaling of memory bandwidth. Overall, due to the scalable nature of matrix profile computation, the use of HPC resources has proven to be a game-changer for matrix profile computation in practice, offering a high throughput computation for real-world use cases.

## 8.2 Collective Effect of Presented Approaches

We underscore that the improvements in computational approaches and resulting efficiency improvements from the seemingly distinct aspects of matrix profile computation (that were investigated in this work) are naturally synergistic. In fact, they are rather additive and accumulative in their impact. Overall, by expanding our understanding of these interrelated dimensions, we contributed to the evolving landscape of matrix profile computation and its potential applications in HPC environments.

In Chapter 5, we demonstrated how the scaling approach introduced in Chapter 4 can be seamlessly combined with computation on GPUs and with reduced- and [mixed-precision](#) schemes. Although our focus was initially on a small scale, i.e., the single-node

multi-GPU scenario, extrapolating from our scaling results on the SuperMUC-NG system and taking the broader studies on matrix profile scalability into consideration, we anticipate similar favorable scaling behaviors on a larger scale. The tree-based approach, another significant aspect of this work, also exhibits the potential for further optimization when coupled with reduced- and mixed-precision computation. This integration not only can reduce data traffic over high-performance networks but also can alleviate computational demands. We, therefore, anticipate analogous computational efficiency improvements for the tree-based approach when leveraging reduced- and mixed-precision computation. On another front, considering the increasing presence of GPUs in Tier-0 HPC centers opens up new avenues for investigation. Particularly, the exploration of tree-based approaches on systems equipped with GPUs at a larger scale represents an intriguing research direction.

### 8.3 Lessons Learned

In the following, we list a number of lessons we learned and experiences gained through research on and the development of the various aspects of this work.

- **Conceptual and Methodological Development Lessons:** one of the key lessons learned during the development of this thesis is the versatility and effectiveness of the matrix profile concept in the data science community. The concept of the matrix profile has undergone numerous extensions and special flavors, thereby widening its scope of applications. In this context, this work contributes to the vital aspect of the evolution of computational methods for the matrix profile. It includes the capabilities of leveraging large supercomputers, the versatility of hardware used for matrix profile computation, and employing diverse and rich methods and techniques that are common in the HPC world, including approximate methods and tree data structures. Additionally, this work helped to employ reduced- and mixed-precision methods and hardware support, which is already commonplace in machine-learning-driven applications but not in matrix profile computation. These evolutions help to make the matrix profile concept more practical and robust for real-world applications.
- **Technological Aspects Lessons:** from the technological point of view, a variety of hardware resources have been exploited for the computation of the matrix profile. Our work specifically contributes to leveraging various hardware, including CPUs, GPUs, and even FPGAs<sup>1</sup>. Furthermore, we observe that analytics workloads are increasingly deployed in the cloud environments to take advantage of the ease of access, ease of provisioning, and elasticity, making the cloud environment a natural choice for many data analytics tasks. The cloud-based frameworks like SCAMP have been specifically designed to facilitate matrix profile computation in the cloud environment and take advantage of the points listed above. Additionally,

---

<sup>1</sup>our efforts around leveraging FPGAs for matrix profile computation are not discussed in this work.

special flavors of matrix profile computation methods (e.g., online schemes) exist that are suitable for real-time data stream processing, where cloud computation models (e.g., serverless and FaaS) could become a better fit naturally. However, we learned that time-series mining and, specifically, matrix profile computation are computationally demanding tasks, which is a clear distinction from common big data analytic workloads. Therefore, there is a natural need for leveraging the abundance of resources scaling the computation in **HPC**. It is important to note that High-Performance Computing (**HPC**) systems still hold a significant role in this realm. This remains particularly valid when dealing with explorative time series mining tasks that involve large datasets. The reason behind this suitability lies in the nature of these workloads. Such workloads often involve batch processing, where the mining of massive datasets is performed in bulk. **HPC** systems with their parallel processing capabilities and dedicated hardware, combined with the approaches leveraging systems (like the ones introduced in this work) excel in handling such workloads efficiently.

- **Limitations and Practical Implications:** While the methods and evaluations presented in this work target matrix profile computation, the developed approaches can be applied to other fields. Similar scaling methods and optimizations, as well as **GPU** computation, can be applied to data mining with different data types. Furthermore, the presented pipelining and forest mechanisms can be applied to applications with similar scaling bottlenecks. Also, these mechanisms can easily be plugged into any tree-based method, e.g., generic tree-based nearest neighbor solvers, to improve scaling efficiency.

## 9 Outlook and Conclusion

In this work, we presented new methods and schemes for the scalable offline analysis and mining of time series data. In particular, we focused on multi- and single-dimensional matrix profile computation across various HPC platforms, specifically targeting CPUs to provide the bulk computational power as well as GPU for acceleration. Our innovations can be summarized across three main thrusts:

**Scalability on HPC Systems:** we pioneered a scalable prototype for matrix profile computation,  $(MP)^N$ , that leverages CPU-based HPC systems and targets large-scale multi-dimensional time series analysis. We demonstrated that matrix profile computations could scale up to 256K cores on large-scale HPC systems, in particular SuperMUC-NG. This capability significantly enhances the matrix profile approach’s applicability to large-scale, real-world problems, achieving unprecedented kernel performance of 1.3 petaflop/s.

**GPU Acceleration and Reduced- and Mixed-Precision:** we introduced the first GPU-based scheme specifically designed for accelerating multi-dimensional matrix profile computations. Our optimized data layout, reduced- and mixed-precision modes, and novel tiling scheme capitalize on GPU hardware features to achieve high performance and accuracy.

**Approximate Parallel Schemes for HPC:** finally, we addressed the computation of matrix profiles by developing an approximate parallel iterative tree-based approach, also targeting CPU-based HPC systems. We introduced and leveraged optimization schemes mitigating the scaling issues to improve the parallel efficiency by a factor of two on 32K cores, enabling the computation of large time series matrix profiles that were previously infeasible in just a matter of minutes.

The strides we’ve made in matrix profile computation and time series analysis form a solid ground for future work. Future research can aim at developing a unified framework that evaluates various matrix profile methods not just on accuracy and performance but on a range of holistic metrics like energy consumption, memory footprint, and robustness. As computational solutions become more complex, their energy consumption needs are rising. Investigating the energy consumption of these methods alongside traditional performance metrics can offer a better picture of their operational efficacy. Moreover, the investigation of various computational models and paradigms is highly interesting. In our work, we mainly adopted the native programming models; however, investigation of models with higher levels of abstraction can be very beneficial as most users tend to rely on such models growingly. The development of libraries that leverage hardware-specific features for these new methods can make it easier for practitioners to take advantage of these advances without needing to understand the underlying hardware intricacies. At the same time, the use of new special accelerators, such as wafer scale processors

## 9 Outlook and Conclusion

that allow leveraging large amounts of on-chip memory bandwidth, is a viable option for future research.

Despite these exciting avenues for research, the presented approaches have been evaluated practically and extensively, and multiple real-world examples and scenarios for applying the methods are demonstrated. These methods substantially reduce the computation time and efficiency in the computation of matrix profiles. These works coalesce to demonstrate that matrix profiles can be computed at an unprecedented scale and speed without sacrificing accuracy, courtesy of tailored algorithmic advances and hardware-specific optimizations. Overall, these methods are geared towards more efficient large-scale High-Performance Data Analytics ([HPDA](#)) leveraging [HPC](#) for time series analysis, representing a huge leap in the feasibility of applying matrix profile techniques for large-scale data mining and setting new benchmarks for the state-of-the-art.



# Appendices



# A1 List of Publication

This appendix lists the author's publications, divided into two sections: those that directly contributed to the development of this dissertation and those that provided indirect or distant contributions.

## A1.1 Works with Direct Association to this Dissertation

- Amir Raoofy, Roman Karlstetter, Martin Schreiber, Carsten Trinitis, Martin Schulz; Overcoming Weak Scaling Challenges in Tree-based Nearest Neighbor Time Series Mining; ISC High Performance 2023.
- Yi Ju, Amir Raoofy, Dai Yang, Erwin Laure, Martin Schulz; Exploiting Reduced Precision for GPU-based Time Series Mining; IPDPS 2022.
- Roman Karlstetter, Amir Raoofy, Martin Radev, Carsten Trinitis, Jakob Hermann, and Martin Schulz; Living on the Edge: Efficient Handling of Large Scale Sensor Data; CCGrid 2021.
- Amir Raoofy, Roman Karlstetter, Dai Yang, Carsten Trinitis, Martin Schulz; Time Series Mining at Petascale Performance; ISC High Performance 2020 (Winner of the Hans Meuer Best Paper Award).
- Roman Karlstetter, Robert Widhopf-Fenk, Jakob Hermann, Driek Rouwenhorst, Amir Raoofy, Carsten Trinitis, Martin Schulz; Turning Dynamic Sensor Measurements From Gas Turbines Into Insights: A Big Data Approach; ASME Turbo-Expo conference 2019.
- Amir Raoofy, Bengisu Elis, Vincent Bode, Minh Thanh Chung, Sergej Breiter, Maron Schlemmon, Dennis-Florian Herr, Karl Fuerlinger, Martin Schulz, Josef Weidendorfer; The BEAST LAB: A Practical Course on Experimental Evaluation of Diverse Modern HPC Architectures and Accelerators; Tenth Workshop on Best Practices for HPC Training and Education, SC'23 Workshop.

## A1.2 Other Works

- Mohak Chadha, Eishi Arima, Amir Raoofy, Michael Gerndt, Martin Schulz; Sustainability in HPC: Vision and Opportunities; Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis.

## A1 List of Publication

- Julian Scheipl, Amir Raoofy, Michael Ott, Josef Weidendorfer; Phase-aware System-Side Sampling for [HPC](#); CF '23: Proceedings of the 20th ACM International Conference on Computing Frontiers.
- Andreas Koch, Gabriel Dax, Michael Petry, Harvey Gomez, Amir Raoofy, Urvij Saroliya, Max Ghiglione, Gianluca Furano, Martin Werner, Carsten Trinitis, Martin Langer; Reference Implementations for Machine Learning Application Benchmark; First European Data Handling & Data Processing Conference – EDHPC 2023.
- Andreas Koch, Michael Petry, Max Ghiglione, Amir Raoofy, Gabriel Dax, Gianluca Furano, Martin Werner, Carsten Trinitis, Martin Langer; Machine Learning Application Benchmark; Proceedings of the 20th ACM International Conference on Computing Frontiers.
- Amir Raoofy, Josef Weidendorfer, Michael Ott; Always-On Instrumentation for Application Introspection in [HPC](#); Proceedings of the 19th ACM International Conference on Computing Frontiers.
- Max Ghiglione, Vittorio Serra, Amir Raoofy, Gabriel Dax, Carsten Trinitis, Martin Werner, Martin Schulz, Gianluca Furano; Survey of Frameworks for Inference of Neural Networks in Space Data Systems; Data Systems in Aerospace (DASIA). Eurospace.
- Amir Raoofy, Gabriel Dax, Vittorio Serra, Max Ghiglione, Martin Werner, Carsten Trinitis; Benchmarking and Feasibility Aspects of Machine Learning in Space Systems; Proceedings of the 19th ACM International Conference on Computing Frontiers.
- Max Ghiglione, Amir Raoofy, Gabriel Dax, Gianluca Furano, Richard Wiest, Carsten Trinitis, Martin Werner, Martin Schulz, Martin Langer; Machine Learning Application Benchmark for In-Orbit On-Board Data Processing; OBDP 2021.
- Amir Raoofy, Gabriel Dax, Max Ghiglione, Martin Langer, Carsten Trinitis, Martin Werner, and Martin Schulz; Benchmarking Machine Learning Inference in [FPGA](#)-based Accelerated Space Applications; MLBench 2021.
- Amir Raoofy, Dai Yang, Josef Weidendorfer, Carsten Trinitis and Martin Schulz; Enabling Malleability for Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics using LAIK; PARS Workshop 2019.
- Arash Bakhtiari, Dhairya Malhotra, Amir Raoofy, Miriam Mehl, Hans-Joachim Bungartz, George Biros; A Parallel Arbitrary-Order Accurate AMR Algorithm for the Scalar Advection-Diffusion Equation; SC '16.

## A2 Complexity of Tree-based Approach, with Pipelining and Forest Optimization

To better understand the performance characteristics of the tree-based method, we reuse the complexity analysis of Xiao et al. [XB16] and adapt it according to the pipelining and forest mechanisms.

For this complexity analysis, the following assumptions and simplifications are made: (1) We assume that the size of the reference and query sets are equal (i.e.,  $m=n$ ). (2) We assume that the tree data structure is fully parallel and the depth of the tree is equal to  $\log(p)$ .

We split the time for the execution of a single iteration into smaller chunks (Equation (1)) based on the phases that we introduced in Section 6.5.

$$T_{Iteration} = T_{Phase_1} + T_{Phase_2} + T_{Phase_3} \quad (A2.1)$$

$$T_{Phase_1} = T_{transformation} + T_{broadcast} = \mathcal{O}\left(n \cdot d/(p/e)\right) + \mathcal{O}\left(\log(p/e)\right) \quad (A2.2)$$

$$T_{Phase_2} = T_{max\_variance\_reduce} + T_{split\_redistribute\_balance} = \mathcal{O}\left(\log(p/e) \cdot \log(n) + n/(p/e)\right) + \mathcal{O}\left(\log(n) \cdot \log^2(p/e)\right) + \mathcal{O}\left(n \cdot d/(p/e)\right) + \mathcal{O}\left(n \cdot d \cdot \log(p/e)/(p/e)\right) \quad (A2.3)$$

$$T_{Phase_3} = T_{solve\_NNJ} + T_{merge} = \mathcal{O}\left(n_c^2 \cdot d\right) + \mathcal{O}\left(n \cdot d \log(p/e)/(p/e)\right) + \mathcal{O}\left(p/e\right) \quad (A2.4)$$

The mathematical terms that are highlighted in red are removed by the optimizations proposed in Chapter 6.

## A3 Billion Scale Experiment using Tree Approach

To showcase the capabilities of the tree-based approach, we run a matrix profile computation on a billion-record random walk dataset ( $n = 1$  billion) with a window size  $m = 128$ , using  $48K$  cores of the [SuperMUC-NG](#) to 99% in 19 minutes. This run reaches  $\approx 338$  teraflop/s maximum kernel performance. There are only two previous works that conduct experiments at this scale: 1) in 2019, Zimmerman et al. [[ZKS<sup>+</sup>19](#)] performed matrix profile computation for a time series with 1-billion-record on AWS spot instances using the [SCAMP](#) algorithm, which took between 10.3 hours to 2.5 days. Compared to that, the tree-based approach reaches one to two orders of magnitude faster speed. 2) in 2022, Lu et al. [[LWM<sup>+</sup>22](#)] conducted anomaly detection using the [DAMP](#) algorithm on one-billion-record in around one hour on a commodity desktop system in a single run. While the tree-based method is still 3x faster, it is highly inefficient compared to [DAMP](#). However, the matrix profile computed with the tree-based method can be applied to various data mining and machine learning tasks, while [DAMP](#) only targets anomaly detection.

## A4 Link to Prototype Codes

The prototype codes developed for this work are available on GitHub in the following repositories! For access, contact the author.

- <https://github.com/amir-raoofy/mpn>
- <https://github.com/amir-raoofy/mpngpu>
- <https://github.com/amir-raoofy/kdmp>

# A5 System and Hardware Specifications

This appendix provides an overview and pointers to the specifications of the systems and hardware used in this work.

## A5.1 NVIDIA GPUs

In this Appendix, we list the specification of NVIDIA GPUs are utilized in our work. The specs, summarized in Figure A5.1, cover various aspects, including their computing resources, memory capacity and bandwidth, and energy consumption. The studies in this work all use PCIe-based variation of the NVIDIA GPUs.

	Tesla V100 PCIe	Tesla V100 SXM2		NVIDIA A100 SXM4 for NVIDIA HGX**	NVIDIA A100 PCIe GPU		Form Factor	H100 SXM	H100 PCIe
GPU Architecture	NVIDIA Volta		GPU Architecture	NVIDIA Ampere			FP64	34 teraFLOPS	26 teraFLOPS
NVIDIA Tensor Cores	640		Double-Precision Performance	FP64: 9.7 TFLOPS			FP64 Tensor Core	67 teraFLOPS	51 teraFLOPS
NVIDIA CUDA® Cores	5,120		Single-Precision Performance	FP32: 19.5 TFLOPS Tensor Float 32 (TF32): 156 TFLOPS   312 TFLOPS*			FP32	67 teraFLOPS	51 teraFLOPS
Double-Precision Performance	7 TFLOPS	7.5 TFLOPS	Half-Precision Performance	312 TFLOPS   624 TFLOPS*			TF32 Tensor Core	989 teraFLOPS*	756teraFLOPS*
Single-Precision Performance	14 TFLOPS	15 TFLOPS	Bfloat16	312 TFLOPS   624 TFLOPS*			BFLOAT16 Tensor Core	1,979 teraFLOPS*	1,513 teraFLOPS*
Tensor Performance	112 TFLOPS	120 TFLOPS	Integer Performance	INT8: 624 TOPS   1,248 TOPS* INT4: 1,248 TOPS   2,496 TOPS*			FP16 Tensor Core	1,979 teraFLOPS*	1,513 teraFLOPS*
GPU Memory	16 GB HBM2		GPU Memory	40 GB HBM2			FP8 Tensor Core	3,958 teraFLOPS*	3,026 teraFLOPS*
Memory Bandwidth	900 GB/sec		Memory Bandwidth	1.6 TB/sec			INT8 Tensor Core	3,958 TOPS*	3,026 TOPS*
ECC	Yes		Error-Correcting Code	Yes			GPU memory	80GB	80GB
Interconnect Bandwidth*	32 GB/sec	300 GB/sec	Interconnect Interface	PCIe Gen4: 64 GB/sec Third generation NVIDIA® NVLink®: 480 GB/sec**	PCIe Gen4: 64 GB/sec Third generation NVIDIA® NVLink®: 600 GB/sec**		GPU memory bandwidth	3.35TB/s	2TB/s
System Interface	PCIe Gen3	NVIDIA NVLink	Form Factor	4/8 SXM GPUs in NVIDIA HGX® A100	PCIe		Decoders	7 NVDEC 7 JPES	7 NVDEC 7 JPES
Form Factor	PCIe Full Height/Length	SXM2	Multi-Instance GPU (MIG)	Up to 7 GPU instances			Max thermal design power (TDP)	Up to 700W (configurable)	300-350W (configurable)
Max Power Consumption	250 W	300 W	Max Power Consumption	400 W	250 W		Multi-Instance GPUs	Up to 7 MIGs @ 10GB each	
Thermal Solution	Passive		Delivered Performance for Top Apps	100%	90%		Form factor	SXM	PCIe Dual-slot air-cooled
Compute APIs	CUDA, DirectCompute, OpenCL™, OpenACC		Thermal Solution	Passive			Interconnect	NVLink: 900GB/s PCIe Gen5: 128GB/s	NVLink: 600GB/s PCIe Gen5: 128GB/s
			Compute APIs	CUDA®, DirectCompute, OpenCL™, OpenACC®			Server options	NVIDIA HGX® H100 Partner and NVIDIA-Certified Systems with 4 or 8 GPUs NVIDIA DGX® H100 with 8 GPUs	Partner and NVIDIA-Certified Systems with 1-8 GPUs
							NVIDIA AI Enterprise	Add-on	Included

Figure A5.1: Specification of NVIDIA V100 (left), NVIDIA A100 (middle), and NVIDIA H100 (right) GPUs.

## A5.2 Systems and Supercomputers

We list the following links for accessing the specifications of supercomputers used in this work.

- SuperMUC-NG: <https://doku.lrz.de/hardware-of-supermuc-ng-phase-1-11482553.html>



- MPCDF Raven: <https://docs.mpcdf.mpg.de/doc/computing/raven-details.html>
- NVIDIA Selene: <https://www.nvidia.com/en-us/on-demand/session/supercomputing2020-sc2019/>
- LRZ BEAST: No public specification is available. We only rely on the press page: [https://www.lrz.de/presse/ereignisse/2020-11-06\\_BEAST/2020-11-06\\_BEAST/](https://www.lrz.de/presse/ereignisse/2020-11-06_BEAST/2020-11-06_BEAST/)

# **Acronyms and Abbreviations**



*Acronyms and Abbreviations*

- AI** Artificial Intelligence. 6, 8, 43, 45, 49
- AMG** Algebraic Multigrid. 97, 125
- ARIMA** Autoregressive Integrated Moving Average. 4, 22
- ASTRO** Celestial Objects Dataset. 126, 127
- AVX512** 512-bit Advanced Vector Extension. 41
- BAdW** Bayerische Akademie der Wissenschaften. 8
- BFLOAT16** 16-bit Brain Floating Point Format. 16, 99
- BFS** Bayerische Forschungstiftung. x
- BLAS** Basic Linear Algebra Subprograms. 44, 108
- CDU** Cooling Distribution Units. 21
- CPU** Central Processing Unit. iii–vi, 9, 11, 16, 41–45, 47–50, 52, 62, 63, 65, 67, 78, 81, 82, 84–86, 89, 91, 92, 94, 96, 98, 101, 115, 121, 133, 135
- CU** Compute Unit. 49
- CUDA** Compute Unified Device Architecture. 44, 47–49, 52, 86, 88, 98
- CXL** Compute Express Link. 42
- DAMP** Discord Aware Matrix Profile. 52, 53, 101, 142
- DBMS** Database Management Systems. 23
- DCDB** Datacenter Database. 21
- DEEP** Dynamical Exascale Entry Platform - Extreme Scale Technologies. 20
- DGEMM** Double-precision General Matrix Multiply. 108, 109, 115
- DNA** Deoxyribonucleic Acid. 7
- DRAM** Dynamic Random Access Memory. 92, 93
- DTW** Dynamic Time Warping. 23
- ECG** Electrocardiogram SignalDataset. 126, 127
- ED** Euclidean Distance. 23
- EMG** Electromyography Signal Dataset. 126, 127

- ESA** European Space Agency. [x](#)
- EU-JU** European High-Performance Computing Joint Undertaking. [x](#)
- EXE** Execution. [41](#)
- FA-LAMP** FPGA-Accelerated Learned Approximate Matrix Profile. [52](#)
- FFT** Fast Fourier Transform. [52](#)
- FLANN** A Fast Library for Approximate Nearest Neighbors. [104](#)
- flop/s** Floating Point Operations Per Second. [6](#), [43](#), [46](#), [72](#)
- FP16** 16-bit Floating Point. [16](#), [49](#), [87](#), [88](#), [92–94](#), [96–99](#), [124](#)
- FP16C** Floating Point 16 with Compensated Prefix Sum. [87](#), [88](#), [93](#), [94](#), [96–98](#), [124](#), [125](#)
- FP32** 32-bit Floating Point. [16](#), [49](#), [87](#), [88](#), [92–94](#), [124](#)
- FP64** 64-bit Floating Point. [16](#), [49](#), [87–90](#), [92–94](#), [124](#)
- FP8** 8-bit Floating Point. [16](#)
- FP80** 80-bit Floating Point. [16](#)
- FPGA** Field-Programmable Gate Array. [45](#), [52](#), [54](#), [133](#), [140](#)
- GAP** Power Consumption for households Dataset. [126](#), [127](#)
- GASPI** Global Address Space Programming Interface. [47](#)
- GCS** Gauss Centre for Supercomputing. [x](#)
- GDDR** Graphics Double Data Rate. [49](#)
- GNU** Gnu’s Not Unix. [47](#)
- GPFS** General Parallel File System. [43](#)
- GPU** Graphics Processing Unit. [iii–vi](#), [6–10](#), [41–45](#), [47–50](#), [52](#), [53](#), [66](#), [70](#), [79–94](#), [96–100](#), [103](#), [122–125](#), [128](#), [131–135](#), [139](#), [144](#)
- GT** Gas Turbine. [123](#), [124](#), [126](#), [127](#)
- HBM** High Bandwidth Memory. [10](#), [80](#)
- HIP** Heterogeneous-compute Interface for Portability. [44](#), [47](#)
- HLS** High Level Synthese. [52](#)

## *Acronyms and Abbreviations*

- HPC** High Performance Computing. [iii–vi](#), [4](#), [6–12](#), [17](#), [18](#), [20–22](#), [30](#), [42–53](#), [57–60](#), [62–72](#), [74](#), [76–79](#), [81](#), [84](#), [88](#), [97](#), [100–105](#), [107](#), [110](#), [112](#), [115](#), [121](#), [122](#), [124–126](#), [131–136](#), [139](#), [140](#)
- HPC-ODA** HPC Monitoring Dataset. [97](#), [124–127](#)
- HPDA** High Performance Data Analytics. [81](#), [87](#), [99](#), [136](#)
- HPE** Hewlett Packard Enterprise. [44](#)
- HPL** High-Performance Linpack. [6](#), [21](#), [97](#), [125](#)
- HZ** Hertz. [14](#), [15](#), [97](#)
- I/O** Input/Output. [17](#), [43](#), [46](#), [58](#), [59](#), [68–71](#), [73](#), [77](#), [80](#)
- IC** Integrated Circuit. [5](#)
- ID** Instruction Decode. [41](#)
- IEEE** Institute of Electrical and Electronics Engineers. [16](#)
- IF** Instruction Fetch. [41](#)
- InsectEPG** Insect Behaviours Dataset. [126](#), [127](#)
- INT16** 16-bit Integer. [16](#)
- INT32** 32-bit Integer. [16](#)
- INT64** 64-bit Integer. [16](#)
- INT8** 8-bit Integer. [16](#)
- IOP** I/O Operations. [70](#)
- IPU** Intelligence Processing Unit. [45](#)
- ITAC** Intel Trace Analyzer, and Collector. [47](#)
- L1/TEX** Level 1 Texture Cache. [92](#), [93](#)
- LAMP** Learned Approximate Matrix Profile. [52–54](#), [101](#), [103](#)
- LDMS** Lightweight Distributed Metric Service. [21](#)
- LIKWID** Like I Knew What I’m Doing. [47](#), [72](#), [77](#)
- MEM** Memory access. [41](#)
- MGAB** Synthetic Anomalies Time Series Dataset. [126](#), [127](#)

- MIMD** Multiple Instruction Streams, Multiple Data Streams. [42](#), [43](#), [48](#)
- MIR** Music Information Retrieval. [7](#)
- MISD** Multiple Instruction Streams, Single Data Stream. [42](#), [43](#)
- MKL** Intel Math Kernel Library. [115](#)
- ML** Machine Learning. [6](#), [87](#)
- MP-MPI** MPI-enabled Matrix Profile. [52](#), [53](#), [63](#)
- MPI** Message Passing Interface. [44](#), [47](#), [48](#), [53](#), [58](#), [59](#), [68–70](#), [72–74](#), [76](#), [77](#), [110](#), [112](#), [113](#), [115](#), [120](#)
- MPIP** A Lightweight Message Passing Interface Profiler. [47](#)
- mSTAMP** Multi-dimensional Scalable Time Series Anytime Matrix Profile. [52](#), [53](#), [57–59](#), [100](#)
- MW** Megawatt. [14](#)
- NN** Nearest Neighbor. [108](#)
- NUMA** Non-Uniform Memory Access. [73](#)
- OpenMP** Open Multi-Processing. [44](#), [47](#), [51](#), [77](#)
- PANDA** Extreme Scale Parallel K-Nearest Neighbor on Distributed Architectures. [104](#)
- PAPI** Performance Application Programming Interface. [47](#)
- PC** Pearson’s Correlation. [23](#), [38](#)
- PCIe** Peripheral Component Interconnect Express. [42](#), [144](#)
- PDU** Power Distribution Units. [21](#)
- PFS** Parallel File Systems. [43](#)
- PGAS** Partitioned Global Address Space. [47](#)
- PMUs** Performance Monitoring Units. [47](#)
- PreSCRIMP** Pre-Subsequence Correlation-based Time Series Matrix Profile. [52](#), [53](#), [103](#), [126](#), [127](#)
- PSML** Energy Grid Dataset. [126](#), [127](#)
- RKDT** Randomized KD Tree. [104](#)

*Acronyms and Abbreviations*

- RPM** Revolutions Per Minute. [14](#)
- RQ** Research Questions. [58](#), [81](#), [102](#)
- SCAMP** SCAlable Matrix Profile. [52](#), [53](#), [63](#), [103](#), [133](#), [142](#)
- SCRIMP** Subsequence Correlation-based Time Series Matrix Profile. [103](#), [104](#)
- SCRIMP++** Subsequence Correlation-based Time Series Matrix Profile. [52–54](#), [101](#), [103](#), [105](#), [106](#), [114–116](#), [126](#), [127](#)
- ScrimpCo** Matrix Profile on Commodity Processors. [52](#), [53](#)
- SIMD** Single Instruction Stream, Multiple Data Streams. [41](#), [42](#)
- SIMT** Single Instruction Multiple Thread. [42](#), [49](#)
- SISD** Single Instruction Stream, Single Data Stream. [42](#)
- SLES** SUSE Linux Enterprise Server Distribution. [44](#)
- SLURM** Simple Linux Utility for Resource Management. [44](#)
- SmartNIC** Smart Network Interface Card. [45](#)
- SMs** Streaming Multiprocessors. [49](#), [82](#), [90](#)
- SMT** Simultaneous Multithreading. [42](#)
- STAMP** Scalable Time Series Anytime Matrix Profile. [52](#)
- STOMP** Scalable Time Series Ordered-search Matrix Profile. [36](#), [37](#), [39](#), [40](#), [52–54](#), [58–60](#), [91](#), [100](#), [101](#), [105](#)
- SVE** Scalable Vector Extension. [42](#)
- TAU** Tuning and Analysis Utilities. [47](#)
- TF32** TensorFloat-32. [16](#), [49](#), [99](#)
- TPUs** Tensor Processing Units. [45](#)
- WB** Writeback. [41](#)



# Glossary



- A64FX** The Fujitsu-designed 64-bit ARM microprocessor is employed in the Fugaku system, a supercomputer developed collaboratively by RIKEN and Fujitsu. This microprocessor powers up the Fugaku supercomputer. [42](#), [44](#)
- Alveo** AMD Xilinx Alveo FPGAs is a PCIe-based FPGA card targeting data center and cloud. [54](#)
- AMD Instinct** AMD Instinct is AMD's brand of data center GPUs.. [6](#), [44](#)
- Barnes-Hut** The Barnes-Hut algorithm is an efficient computational method for simulating the motion of large systems of particles, like galaxies, by approximating the influence of groups of nearby particles as a single entity. [104](#)
- C++** C++ is a high-level programming language known for its versatility and efficiency. It was developed as an extension of the C language to include object-oriented features like classes and inheritance, which help in organizing and managing complex programs.. [52](#), [72](#), [88](#), [115](#)
- CAPS** Chair of Computer Architecture and Parallel Systems at technological University of Munich. [ix](#)
- Cerebras** Cerebras Systems is an American artificial intelligence company. Cerebras builds computer systems including its special purpose Wafer Scale Engine (WSE) which is an AI chip targeting intelligence deep learning applications. [45](#)
- CORAL** CORAL stands for Collaboration of Oak Ridge, Argonne, and Lawrence Livermore. It is a consortium is a joint initiative involving three of the United States' most prominent national laboratories: Oak Ridge National Laboratory (ORNL), Argonne National Laboratory (ANL), and Lawrence Livermore National Laboratory (LLNL). The primary objective of this consortium is to accelerate the development and deployment of high-performance computing (HPC) systems in the United States.. [21](#), [30](#), [125](#)
- DGX-1** The DGX-1, built by NVIDIA, is an integrated system tailored for deep learning and AI tasks. It is powered by multiple Tesla V100 GPUs with Tensor Cores.. [89](#), [92](#), [93](#)
- EPYC** EPYC is a brand of high-performance x86-64 server processors designed and manufactured by Advanced Micro Devices (AMD). [42](#), [44](#)
- exaflop/s** One quintillion floating point operations per second.. [6](#)
- exascale** Exascale systems refers to computing systems capable of calculating at least one quintillion IEEE-754 double precision (64-bit) operations per second.. [6](#), [20](#), [44](#), [100](#)

**Flynn's Taxonomy** Flynn's Taxonomy, introduced by Michael J. Flynn in 1966, is a taxonomy for computer systems. [42](#), [48](#)

**FORTRAN** FORTRAN, is a high-level programming language developed by IBM in the 1950s. Primarily designed for numerical and scientific computing. [48](#)

**Frontier** As of December 2023, Frontier is the world's fastest supercomputer, and it is the world's first Exascale supercomputer.. [6](#), [44](#)

**Fugaku** The system Fugaku, established as the flagship High-Performance Computing (HPC) system at RIKEN Center for Computational Science RIKEN, stood the top position in the TOP500 list as of June 2020 until June 2022. Comprising 158,976 compute nodes, the system boasts an aggregate core count of 7,630,848 cores while operating at a power consumption rate of 28 megawatts. At its computational core lies the Fujitsu A64FX CPU architecture. [44](#)

**Google Cloud** Google Cloud Platform is a suite of cloud computing services offered by Google that provides a series of modular cloud services including computing, data storage, data analytics, and machine learning, alongside a set of management tools.. [52](#)

**Gustafson's Law** Gustafson's Law, proposed by John Gustafson in 1988, is a theoretical about weak scaling limits in parallel computing. [7](#), [46](#)

**Hewlett Packard Enterprise** Hewlett Packard Enterprise (HPE) is a multinational information technology company that specializes in providing a wide range of hardware, software, and services. HPE delivers products and solutions for data storage, networking, servers, and cloud computing.. [6](#)

**Icelake** Ice Lake is Intel's codename for its 10th generation Core processors, featuring a 10nm process.. [71](#), [73](#), [74](#)

**KD-tree** KD-trees (K-dimensional trees) are hierarchical structures designed for efficient multidimensional data partitioning and retrieval, commonly employed in spatial applications such as nearest neighbor searches and range queries. [104](#), [107](#), [108](#), [113](#), [114](#)

**Keras** Keras is a popular open-source neural networks API in Python.. [52](#)

**LAPACK** LAPACK, or Linear Algebra PACKage, is a library of routines for solving linear algebra problems, including linear equations, linear least squares problems, eigenvalue problems, and singular value decomposition. LAPACK provides a collection of highly optimized and portable routines written in Fortran and is widely used for numerical computations in scientific computing applications. [44](#)

- LINPCK** LINPACK is a set of Fortran subroutines designed for the analysis and resolution of linear algebra systems. The LINPACK benchmark specifically tackles systems with a predetermined number of operations in 64-bit floating point arithmetic, enabling the measurement of practically achievable Floating Point Operations Per Second (FLOPS) for a High-Performance Computing (HPC) system. The present implementation of this benchmark is known as HPL (Pet+). [6](#)
- LogP** LogP is a theoretical model proposed by Culler and Singh in 1993 for analyzing the performance of parallel algorithms in distributed computing. [47](#)
- LRZ** LRZ stands for Leibniz Supercomputing Centre. It is a high-performance computing (HPC) center located in Garching near Munich, Germany. LRZ provides advanced computing resources, services, and supports for researchers and scientists in various fields, including physics, chemistry, life sciences, and engineering. The center is known for hosting and operating tier-0 supercomputers, such as SuperMUC and SuperMUC-NG.. [x](#), [8](#), [21](#), [115](#)
- LRZ BEAST** BEAST stands for Bavarian Energy, Architecture, and Software Testbed which is an experimental system at Leibniz Supercomputing Centre to enable research on state-of-the-art HPC hardware.. [8](#), [71](#), [73](#), [145](#)
- Matlab** MATLAB is a high-level programming language and numerical computing environment developed by MathWorks.. [52](#)
- Max Planck Computing** Max Planck Computing and Data Facility is a cross-institutional competence centre of the Max Planck Society to support computational and data sciences.. [90](#)
- MI250X** AMD Instinct MI250 is a GPU from Instinct Family Instinct, manufactured by AMD.. [6](#), [44](#)
- Mixed-precision** Mixed precision combines different numerical precisions for solving parts of a problem, like 16-bit and 32-bit floating point formats, to balance computational efficiency and numerical accuracy. This technique is commonly used in applications such as deep learning and scientific computing for faster processing while maintaining acceptable accuracy levels. [iii](#), [iv](#), [9](#), [10](#), [20](#), [46](#), [49](#), [50](#), [54](#), [80](#), [81](#), [87–89](#), [92–97](#), [99](#), [101](#), [123–125](#), [131–133](#), [135](#)
- Moore's law** Moore's Law, as articulated by Gordon E. Moore in 1965 (Moo65), posits that the complexity, measured in terms of the number of components per integrated circuit, has been doubling approximately every year.. [4](#), [5](#)
- NEON** A SIMD extention for Arm systems. [42](#)

- NVIDIA A100** NVIDIA A100 GPU based on the Ampere architecture. It is part of NVIDIA's Tesla GPUs family of GPUs that is designed for data center and high-performance computing as well as artificial intelligence applications. [45](#), [88](#), [90](#), [92–94](#), [98](#), [144](#)
- NVIDIA Ampere GPU** The NVIDIA Ampere GPU refers to the graphics processing units (GPUs) based on NVIDIA's Ampere architecture. This architecture, introduced by NVIDIA, is the successor to the Volta architecture and is designed to provide significant advancements in performance and efficiency for a range of computing tasks, including graphics rendering and artificial intelligence (AI) workloads. [16](#)
- NVIDIA H100** NVIDIA H100 GPU based on the Hopper architecture. It is part of NVIDIA's Tesla GPUs family of GPUs that is designed for data center and high-performance computing as well as artificial intelligence applications. [88](#), [90](#), [144](#)
- NVIDIA Nsight Compute** NVIDIA Nsight Compute is a performance analysis tool for profiling and optimization of NVIDIA GPU. Specifically designed for CUDA applications running on NVIDIA GPUs.. [47](#), [92](#)
- NVIDIA V100** The NVIDIA V100 is a high-performance graphics processing unit (GPU) designed for data center and high-performance computing applications. It features a large number of CUDA cores, tensor cores for deep learning, high memory bandwidth, and NVLink interconnect technology for efficient multi-GPU configurations. [45](#), [88–90](#), [144](#)
- NVLink** NVLink is a wire-based communications protocol for near-range semiconductor communications developed by NVIDIA. [44](#), [45](#)
- ODA** The utilization of techniques focused on enhancing data center operations by analyzing monitoring and log data, aiming to optimize performance and functionality. [20](#), [21](#), [30](#)
- Omni-Path** Omni-Path is a high-performance, low-latency interconnect technology developed by Intel for use in high-performance computing (HPC) environments. [43](#), [44](#), [48](#)
- OneAPI** OneAPI is an open programming model led by Intel to simplify development across diverse computing architectures, including CPUs, GPUs, and FPGAs.. [115](#)
- OpenCL** OpenCL, or Open Computing Language, is an open standard for parallel programming across various platforms, including CPUs, GPUs, and other accelerators.. [52](#)
- OpenSHMEM** OpenSHMEM (Open Symmetric Hierarchical Memory) is a parallel programming interface and communication library for high-performance computing (HPC) systems. OpenSHMEM supports one-sided communication, has a global

address space model, and is designed for portability across different parallel computing platforms. [47](#)

**P2P** P2P stands for "peer-to-peer," describing a decentralized communication model where end-points interact directly without relying on a central server. [48](#)

**petabyte** One quadrillion bytes.. [18, 44](#)

**petaflop/s** One quadrillion floating point operations per second.. [iv, vi, 8, 44, 77, 135](#)

**petascale** Peta systems refers to computing systems capable of calculating at least one quadrillion IEEE-754 double precision (64-bit) operations per second.. [iii, v, 8, 77, 139](#)

**RIKEN** RIKEN is a scientific research institute in Japan. It hosts the Fugaku system at RIKEN Center for Computational Science. [44](#)

**roofline** The roofline model is a visual performance analysis tool in high-performance computing. It locates the operational intensity and throughput on a graph and helps to identify bottlenecks and optimization opportunities. The roofline chart shows a performance ceiling based on hardware limitations. [47, 72](#)

**self-join** A self-join is a relational database operation that combines rows within the same table, treating it as if it were two separate instances of the same table. In the context of time series data, a self-join is performed on a single time series, allowing for comparisons and analyses within the same sequence. Special treatments are often applied to address trivial similarity matches, ensuring meaningful results when identifying patterns or relationships within the time series itself. [24–26, 28, 31, 35–37, 39, 52, 59, 61, 63, 66, 68, 70, 82](#)

**SenseE** SenseE stands for Sensors on the edge. It is a Research project funded by Bayerische Forschungstiftung (2021-2024), cooperation between TUM and IfTA Ingenieurbüro für Thermoakustik GmbH.. [ix](#)

**Skylake** Skylake refers to Intel's 6th generation Core processors. These processors, are built on a 14nm process.. [44, 72–74](#)

**STREAM** STREAM is a benchmark program designed to measure sustainable memory bandwidth and the corresponding computation rate for simple vector kernels. [72](#)

**STUMPY** STUMPY is a Python library that Efficiently Computes the Matrix Profile. [53](#)

**SuperMUC-NG** SuperMUC-NG is the latest flagship supercomputing system hosted at Leibniz Supercomputing Centre, featuring 6480 Lenovo ThinkSystem SD 650 Direct Water Cooled nodes powered by the Intel Skylake Xeon Platinum 8174

## Glossary

**CPU.** The system employs a fat tree topology interconnect, linking 311050 cores, and achieves a theoretical peak performance of 26.7 petaflop/s. With a total main memory capacity of 719 terabytes, SuperMUC-NG was installed in September 2018. [iv](#), [vi](#), [8](#), [21](#), [44](#), [58](#), [72–74](#), [77](#), [90](#), [102](#), [115](#), [117–121](#), [133](#), [135](#), [142](#), [144](#)

**teraflop/s** One trillion floating point operations per second.. [121](#), [142](#)

**TOP500** The TOP500 is a semiannual event that identifies and ranks the 500 fastest supercomputers globally. The ranking is determined based on the High-Performance Linpack (HPL) benchmark.. [5](#), [6](#), [8](#), [44](#)

**TurbO** TurbO stands for Gas Turbine Optimization using Big Data. It was a Research project funded by Bayerische Forschungsförderung (2017–2020), cooperation between TUM and IfTA Ingenieurbüro für Thermoakustik GmbH.. [ix](#), [18](#), [19](#)

**Ultra96** Ultra96 refers to the Ultra96 Arm-based, Xilinx Zynq UltraScale+ MPSoC development board board. It is a product of Avnet, designed in collaboration with Xilinx. [54](#)

**UNIX** Unix is a family of multitasking, multi-user computer operating systems that derive from the original AT&T Unix, whose development started in 1969 at the Bell Labs research center.. [13](#), [16](#)

**US DOE Oak Ridge National Laboratory** The Oak Ridge National Laboratory (ORNL) is a multipurpose research institution located in Oak Ridge, Tennessee, United States. It is managed by the U.S. Department of Energy and is renowned for its contributions to various scientific and technological domains, including materials science, nuclear science, and high-performance computing. [6](#), [44](#)

**WaferScale** WaferScale involves integrating an entire silicon wafer into a single semiconductor device, maximizing computational power and efficiency. [45](#)

**z-normalized** "Z-normalization" refers to transforming a dataset so its mean is zero and standard deviation is one. This involves subtracting the mean from each data point and dividing by the standard deviation. It's used to bring different datasets to a common scale, aiding in comparisons and processing in various models. [27](#), [28](#), [31](#), [33](#), [35](#), [36](#), [39](#), [95](#), [114](#)



# **Bibliography**



- [AAB<sup>+</sup>14] Anthony Agelastos, Benjamin Allan, Jim Brandt, Paul Cassella, Jeremy Enos, Joshi Fullop, Ann Gentile, Steve Monk, Nichamon Naksinehaboon, Jeff Ogden, Mahesh Rajan, Michael Showerman, Joel Stevenson, Narate Taerat, and Tom Tucker. The Lightweight Distributed Metric Service: A Scalable Infrastructure for Continuous Monitoring of Large Scale Computing Systems and Applications. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 154–165, 2014. doi:10.1109/SC.2014.18.
- [ABB<sup>+</sup>99] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.
- [ABB<sup>+</sup>13] Thomas Alrutz, Jan Backhaus, Thomas Brandes, Vanessa End, Thomas Gerhold, Alfred Geiger, Daniel Grünewald, Vincent Heuveline, Jens Jägersküpper, Andreas Knüpfer, Olaf Krzikalla, Edmund Kügeler, Carsten Lojewski, Guy Lonsdale, Ralph Müller-Pfefferkorn, Wolfgang Nagel, Lena Oden, Franz-Josef Pfreundt, Mirko Rahn, Michael Sattler, Mareike Schmidtobreck, Annika Schiller, Christian Simmendinger, Thomas Sodemann, Godehard Sutmann, Henning Weber, and Jan-Philipp Weiss. *GASPI – A Partitioned Global Address Space Programming Interface*, pages 135–136. Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-35893-7\_18.
- [ABF<sup>+</sup>10] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCTOOLKIT: Tools for Performance Analysis of Optimized Parallel Programs. volume 22, pages 685–701, GBR, apr 2010. John Wiley and Sons Ltd. URL: <http://hpctoolkit.org>.
- [Adv] Advanced Micro Devices, Inc. (AMD). HIP: C++ Runtime API and Kernel Language for AMD and NVIDIA GPUs. URL: <https://docs.amd.com/projects/HIP/en/docs-5.3.0/index.html>.
- [Alm] George Almasi. PGAS (Partitioned Global Address Space) Languages. In David Padua, editor, *Encyclopedia of Parallel Computing*, pages 1539–1545. Springer US. doi:10.1007/978-0-387-09766-4\_210.

- [Amd67] Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference, AFIPS '67 (Spring)*, pages 483–485, New York, NY, USA, 1967. Association for Computing Machinery. doi:10.1145/1465482.1465560.
- [AMN<sup>+</sup>98] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. An Optimal Algorithm for Approximate Nearest Neighbor Searching Fixed Dimensions. volume 45, pages 891–923, New York, NY, USA, November 1998. Association for Computing Machinery. doi:10.1145/293347.293348.
- [Bay20] Bayerische Akademie der Wissenschaften. *Leibniz Supercomputing Centre's (LRZ's) Newest Supercomputer, Ranked 13th in TOP500 List of June 2020*, Online, Accessed on October 19, 2020. URL: <https://doku.lrz.de/display/PUBLIC/SuperMUC-NG>.
- [BDD<sup>+</sup>02] L. Susan Blackford, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, Michael Heroux, Linda Kaufman, Andrew Lumsdaine, Antoine Petitet, Roldan Pozo, Karin Remington, and R. Clint Whaley. An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Trans. Math. Softw.*, 28(2):135–151, jun 2002. doi:10.1145/567806.567807.
- [BDG<sup>+</sup>00] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. volume 14, pages 189–204, USA, aug 2000. Sage Publications, Inc. doi:10.1177/109434200001400303.
- [Ben75] Jon Louis Bentley. Multidimensional Binary Search Trees Used for Associative Searching. volume 18, pages 509–517, New York, NY, USA, September 1975. Association for Computing Machinery. doi:10.1145/361002.361007.
- [BFH<sup>+</sup>04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. volume 23, pages 777–786, New York, NY, USA, aug 2004. Association for Computing Machinery. doi:10.1145/1015706.1015800.
- [BGB<sup>+</sup>16] David Boehme, Todd Gamblin, David Beckingsale, Peer-Timo Bremer, Alfredo Gimenez, Matthew LeGendre, Olga Pearce, and Martin Schulz. Caliper: Performance Introspection for HPC Software Stacks. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 550–560, 2016. doi:10.1109/SC.2016.46.

- [BGMS98] Satish Balay, William Gropp, Lois Curfman McInnes, and Barry F Smith. PETSc, the Portable, Extensible Toolkit for Scientific Computation. volume 2, 1998. URL: <https://petsc.org/>, doi:10.2172/2205494.
- [BH86] Josh Barnes and Piet Hut. A Hierarchical  $O(N \log N)$  Force-Calculation Algorithm. volume 324, pages 446–449, 1986. doi:10.1038/324446a0.
- [BH13] Alice Berard and Georges Hebrail. Searching Time Series with Hadoop in an Electric Power Company. In *Proceedings of the 2nd International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications*, BigMine '13, pages 15–22, New York, NY, USA, 2013. ACM. doi:10.1145/2501221.2501224.
- [BJ70] G.E.P. Box and G.M. Jenkins. Time Series Analysis: Forecasting and Control. Holden-Day series in time series analysis and digital processing. Holden-Day, 1970. URL: <https://books.google.de/books?id=5BVfnXaq03oC>.
- [Bra05] Peter Braam. The Lustre Storage Architecture. 2005. arXiv:1903.01955.
- [Bra17] Berenger Bramas. A Novel Hybrid Quicksort Algorithm Vectorized using AVX-512 on Intel Skylake. volume 8. The Science and Information Organization, 2017. doi:10.14569/IJACSA.2017.081044.
- [CCGV19] Siegfried Cools, Jeffrey Cornelis, P. Ghysels, and Wim Vanroose. Improving Strong Scaling of the Conjugate Gradient Method for Solving Large Linear Systems Using Global Reduction Pipelining. volume abs/1905.06850, 2019. arXiv:1905.06850.
- [CCP<sup>+</sup>10] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. Introducing openshmem: Shmem for the pgas community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/2020373.2020375.
- [CDK<sup>+</sup>01] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. *Parallel Programming in OpenMP*. Morgan kaufmann, 2001.
- [CEF<sup>+</sup>] Soumen Chakrabarti, Martin Ester, Usama Fayyad, Johannes Gehrke, Jiawei Han, Shinichi Morishita, Gregory Piatetsky-Shapiro, and Wei Wang. Data Mining Curriculum: A Proposal (Version 1.0). volume 140. URL: [https://kdd.org/exploration\\_files/CURMay06.pdf](https://kdd.org/exploration_files/CURMay06.pdf).
- [Cen23] Leibniz Supercomputing Centre. DGX-1 V100 at LRZ AI Systems, Online, Accessed on December 16, 2023. URL: <https://doku.lrz.de/display/PUBLIC/LRZ+AI+Systems>.

- [Cha04] Chris Chatfield. *The Analysis of Time Series: An Introduction*. CRC Press, Florida, US, 6th edition, 2004.
- [CHP21] Fan Cheng, Rob J. Hyndman, and Anastasios Panagiotelis. Manifold Learning With Approximate Nearest Neighbors. *ArXiv*, abs/2103.11773, 2021. URL: <https://www.monash.edu/business/ebs/research/publications/ebs/wp03-2021.pdf>.
- [CKP<sup>+</sup>93] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '93, pages 1–12, New York, NY, USA, 1993. Association for Computing Machinery. doi:10.1145/155332.155333.
- [Cla83] K. L. Clarkson. Fast Algorithms for the All Nearest Neighbors Problem. In *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*, pages 226–232, 1983. doi:10.1109/sfcs.1983.16.
- [Col13] Liane Colonna. A Taxonomy and Classification of Data Mining. Semantic Scholar, Online, Accessed on December 16, 2013. URL: <https://api.semanticscholar.org/CorpusID:195444710>.
- [CSG98] David Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.
- [Cur15] Ryan R. Curtin. Faster Dual-Tree Traversal for Nearest Neighbor Search. In Giuseppe Amato, Richard Connor, Fabrizio Falchi, and Claudio Genaro, editors, *Similarity Search and Applications*, pages 77–89, Cham, 2015. Springer International Publishing.
- [Dau23] Eric Daub. Course Notes for Data Analysis in Geophysics 1.0. Online, Accessed on December 16, 2023. URL: [http://www.ceri.memphis.edu/people/egdaub/datanotes/\\_build/html/sac5.html](http://www.ceri.memphis.edu/people/egdaub/datanotes/_build/html/sac5.html).
- [DBp23] DBpedia. About: Z3 (computer). DBpedia Website, Online, Accessed on December 16, 2023. URL: [https://dbpedia.org/page/Z3\\_\(computer\)](https://dbpedia.org/page/Z3_(computer)).
- [DK17] Hoang Anh Dau and Eamonn Keogh. Matrix Profile V: A Generic Technique to Incorporate Domain Knowledge into Motif Discovery. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, pages 125–134, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3097983.3097993.

- [DM98] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry Standard API for Shared-Memory Programming. volume 5, pages 46–55. IEEE, 1998. doi:10.1109/99.660313.
- [DMBS79] Jack J Dongarra, Cleve Barry Moler, James R Bunch, and Gilbert W Stewart. LINPACK users’ guide. SIAM, 1979. doi:10.1137/1.9781611971811.
- [DS21] Fatoumata Dama and Christine Sinoquet. Time Series Analysis and Modeling to Forecast: a Survey. arXiv, 2021. URL: <https://arxiv.org/abs/2104.00164>, doi:10.48550/ARXIV.2104.00164.
- [dW23] Bayerische Akademie der Wissenschaften. Bavarian Academy of Sciences and Humanities — BAdW. <https://badw.de/die-akademie.html>, 2023. Online, Accessed on June 23, 2023. URL: <https://badw.de/die-akademie.html>.
- [EA12] Philippe Esling and Carlos Agon. Time-series Data Mining. volume 45, pages 12:1–12:34, New York, NY, USA, December 2012. ACM. doi:10.1145/2379776.2379788.
- [Eam22] Eamonn Keogh. Electrocardiography Dataset, Online, Accessed on August 15, 2022. URL: [https://www.cs.ucr.edu/~eamonn/ECG\\_one\\_day.zip](https://www.cs.ucr.edu/~eamonn/ECG_one_day.zip).
- [Ech22] Karima Echihabi. Similarity Search for Scalable Data Science: The Past, Present and Exciting Road Ahead. 2022. Online, Accessed on December 16, 2023. URL: <https://wp.sigmod.org/?p=3616>.
- [Eng21] Alexis Friedrich Engelke. Optimizing Performance Using Dynamic Code Generation. 2021. URL: <https://mediatum.ub.tum.de/doc/1614897/1614897.pdf>.
- [Fer19] Ivan Fernandez. SCRIMP FlexFloat Benchmark. GitHub, 2019. URL: <https://github.com/ivanfv/scrimp-flexfloat.git>.
- [Fly] M.J. Flynn. Very High-Speed Computing Systems. volume 54, pages 1901–1909. doi:10.1109/PROC.1966.5273.
- [Fu11] Tak-chung Fu. A Review on Time Series Data Mining. volume 24, pages 164–181, Tarrytown, NY, USA, February 2011. Pergamon Press, Inc. doi:10.1016/j.engappai.2010.09.007.
- [FV17] Amin Fakhrazari and Hamid Vakilzadian. A Survey on Time Series Data Mining. In *2017 IEEE International Conference on Electro Information Technology (EIT)*, pages 476–481, 2017. doi:10.1109/EIT.2017.8053409.
- [GKB<sup>+</sup>23] D. Gouk, M. Kwon, H. Bae, S. Lee, and M. Jung. Memory Pooling With CXL. volume 43, pages 48–57, Los Alamitos, CA, USA, mar 2023. IEEE Computer Society. doi:10.1109/MM.2023.3237491.

- [GKKG03] Ananth Grama, George Karypis, Vipin Kumar, and Anshul Gupta. *Introduction to Parallel Computing*. Addison-Wesley, second edition, 2003.
- [Gmb21a] Forschungszentrum Julich GmbH. Deep - extreme scale technologies. In *Research Project Sponsored by European Commission*, 2017-2021. doi: [10.3030/754304](https://doi.org/10.3030/754304).
- [Gmb21b] Forschungszentrum Julich GmbH. Dynamical Exascale Entry Platform - Software for Exascale Architectures (DEEP-SEA). In *research project sponsored by European Commission*, 2017-2021. URL: <https://www.deep-projects.eu/project.html>.
- [Gra62] John Graunt. Natural and Political Observations Made upon the Bills of Mortality. <http://www.stat.rice.edu/stat/FACULTY/courses/stat431/Graunt.pdf>, 1662. Online, Accessed on June 23, 2023. URL: [https://link.springer.com/chapter/10.1007/978-3-642-81046-6\\_2](https://link.springer.com/chapter/10.1007/978-3-642-81046-6_2).
- [Gus88] John L. Gustafson. Reevaluating Amdahl's Law. volume 31, pages 532–533, New York, NY, USA, may 1988. Association for Computing Machinery. doi: [10.1145/42411.42415](https://doi.org/10.1145/42411.42415).
- [GYD<sup>+</sup>] Shaghayegh Gharghabi, Chin-Chia Michael Yeh, Yifei Ding, Wei Ding, Paul Hibbing, Samuel LaMunion, Andrew Kaplan, Scott E. Crouter, and Eamonn J. Keogh. Domain Agnostic Online Semantic Segmentation for Multi-Dimensional Time Series. In *Data Mining and Knowledge Discovery*. doi: [10.1007/s10618-018-0589-3](https://doi.org/10.1007/s10618-018-0589-3).
- [HHvW<sup>+</sup>20] Stijn Heldens, Pieter Hijma, Ben van Werkhoven, Jason Maassen, Henri Bal, and Rob van Nieuwpoort. Rocket: Efficient and Scalable All-Pairs Computations on Heterogeneous Platforms. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.
- [HP11] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.
- [HSM20] François P. Hamon, Martin Schreiber, and Michael L. Minion. Parallel-in-time multi-level integration of the shallow-water equations on the rotating sphere. volume 407, page 109210, 2020. URL: <https://www.sciencedirect.com/science/article/pii/S0021999119309155>, doi: [10.1016/j.jcp.2019.109210](https://doi.org/10.1016/j.jcp.2019.109210).
- [HZM<sup>+</sup>16] Tian Huang, Yongxin Zhu, Yishu Mao, Xinyang Li, Mengyun Liu, Yafei Wu, Yajun Ha, and Gillian Dobbie. Parallel Discord Discovery. In *Proceedings, Part II, of the 20th Pacific-Asia Conference on Advances in*



- Knowledge Discovery and Data Mining - Volume 9652*, PAKDD 2016, pages 233–244, Berlin, Heidelberg, 2016. Springer-Verlag. URL: [https://doi.org/10.1007/978-3-319-31750-2\\_19](https://doi.org/10.1007/978-3-319-31750-2_19).
- [IBM07] IBM. General Parallel File System. 2007. URL: <https://www.ibm.com/docs/en/gpfs>.
- [IC80] Intel Corporation. x86 Intel 8087 Math Co-Processor, 1980. Online, Accessed on June 23, 2023. URL: [https://pdf.datasheetcatalog.com/datasheets/2300/45014\\_DS.pdf](https://pdf.datasheetcatalog.com/datasheets/2300/45014_DS.pdf).
- [IEE19] IEEE. IEEE Standard for Floating Point Arithmetic. pages 1–84, 2019. doi:10.1109/IEEESTD.2019.8766229.
- [Int] International Organization for Standardization. ISO 8601:1988 - Data Elements and Interchange Formats - Information Interchange - Representation of Dates and Times. Technical report, ISO. URL: <https://www.iso.org/standard/15903.html>.
- [JAV23] JAVIER BLANCO. Time Series Analysis: A Gentle Introduction, Online, Accessed on November 15, 2023. URL: <https://quix.io/blog/time-series-analysis>.
- [JOR11] Peter Wilcox Jones, Andrei Osipov, and Vladimir Rokhlin. Randomized Approximate Nearest Neighbors Algorithm. volume 108, pages 15679–15686. National Academy of Sciences, 2011. doi:10.1073/pnas.1107769108.
- [Jos12] Nicolai M Josuttis. *The C++ Standard Library: A Tutorial And Reference*. Addison-Wesley, 2012.
- [JRS21] Yi Ju, Amir Raoofy, and Martin Schulz. High-performance mining of multidimensional time series with multiple gpus and at reduced precision. In *NVIDIA GTC (GPU Technology Conference)*, 2021. Online, Accessed on June 23, 2023. URL: <https://www.nvidia.com/en-us/on-demand/session/gtcspring21-e31128>.
- [JRY<sup>+</sup>22] Yi Ju, Amir Raoofy, Dai Yang, Erwin Laure, and Martin Schulz. Exploiting Reduced Precision for GPU-based Time Series Mining. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 124–134, 2022. doi:10.1109/IPDPS53621.2022.00021.
- [JYP<sup>+</sup>17] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert

- Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. volume 45, pages 1–12, New York, NY, USA, jun 2017. Association for Computing Machinery. doi:10.1145/3140659.3080246.
- [Kah65] W. Kahan. Pracniques: Further Remarks on Reducing Truncation Errors. volume 8, page 40, New York, NY, USA, January 1965. Association for Computing Machinery. doi:10.1145/363707.363723.
- [Keo02] Eamonn J. Keogh. Exact Indexing of Dynamic Time Warping. In *VLDB*, pages 406–417. Morgan Kaufmann, 2002. URL: <http://dblp.uni-trier.de/db/conf/vldb/vldb2002.html#Keogh02>.
- [Keo23] Eamonn Keogh. Matrix Profile: A Powerful Tool for Time Series Data Mining. Website, Online, Accessed on December 16, 2023. URL: <https://www.cs.ucr.edu/~eamonn/MatrixProfile.html>.
- [Kha20] Paresh Kharya. TensorFloat-32 in the A100 GPU Accelerates AI Training, HPC up to 20x, 2020. Online, Accessed on June 23, 2023. URL: <https://blogs.nvidia.com/blog/2020/05/14/tensorfloat-32-precision-format/>.
- [Kir08] Wilhelm Kirch, editor. *Pearson's Correlation Coefficient*, pages 1090–1091. Springer Netherlands, Dordrecht, 2008. doi:10.1007/978-1-4020-5614-7\_2569.
- [KK] Eamonn Keogh and Shruti Kasetty. On the Need for Time Series Data Mining Benchmarks: A Survey and Empirical Demonstration. volume 7, pages 349–371. doi:10.1023/A:1024988512476.
- [KK93] L.V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA '93*, pages 91–108. ACM Press, September 1993. doi:10.1145/165854.165874.

- [Kle15] Caroline Kleist. Time Series Data Mining Methods. Master’s thesis, Humboldt-Universität zu Berlin, Wirtschaftswissenschaftliche Fakultät, 2015. doi:<http://dx.doi.org/10.18452/14237>.
- [KRH15] Ronan Keryell, Ruyman Reyes, and Lee Howes. Khronos sycl for opencl: A tutorial. In *Proceedings of the 3rd International Workshop on OpenCL, IWOCL ’15*, New York, NY, USA, 2015. Association for Computing Machinery. doi:[10.1145/2791321.2791345](https://doi.org/10.1145/2791321.2791345).
- [KRM<sup>+</sup>12] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir. In Holger Brunst, Matthias S. Müller, Wolfgang E. Nagel, and Michael M. Resch, editors, *Tools for High Performance Computing 2011*, pages 79–91, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
- [KRR<sup>+</sup>21] Roman Karlstetter, Amir Raoofy, Martin Radev, Carsten Trinitis, Jakob Hermann, and Martin Schulz. Living on the Edge: Efficient Handling of Large Scale Sensor Data. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 1–10, 2021. doi:[10.1109/CCGrid51090.2021.00010](https://doi.org/10.1109/CCGrid51090.2021.00010).
- [KWFH<sup>+</sup>19] Roman Karlstetter, Robert Widhopf-Fenk, Jakob Hermann, Driek Rouwenhorst, Amir Raoofy, Carsten Trinitis, and Martin Schulz. Turning Dynamic Sensor Measurements From Gas Turbines Into Insights: A Big Data Approach. In *Proceedings of the ASME Turbo Expo: Power for Land, Sea, and Air*, volume Volume 6: Ceramics; Controls, Diagnostics, and Instrumentation; Education; Manufacturing Materials and Metallurgy, 06 2019. V006T05A021. doi:[10.1115/GT2019-91259](https://doi.org/10.1115/GT2019-91259).
- [KZB22] Amin Kalantar, Zachary Zimmerman, and Philip Brisk. FPGA-Based Acceleration of Time Series Similarity Prediction: From Cloud to Edge. volume 16, New York, NY, USA, dec 2022. Association for Computing Machinery. doi:[10.1145/3555810](https://doi.org/10.1145/3555810).
- [LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *CGO*, pages 75–88, San Jose, CA, USA, Mar 2004. doi:[10.1109/CGO.2004.1281665](https://doi.org/10.1109/CGO.2004.1281665).
- [Lav22] Adam Lavelly. Powering Extreme Scale HPC with Cerebras, 2022. URL: <https://8968533.fs1.hubspotusercontent-na1.net/hubfs/8968533/Powering-Extreme-Scale-HPC-with-Cerebras.pdf>.

- [Law19] Sean M. Law. STUMPY: A Powerful and Scalable Python Library for Time Series Data Mining. *The Journal of Open Source Software*, 4(39):1504, 2019. doi:10.21105/joss.01504.
- [LWM<sup>+</sup>22] Yue Lu, Renjie Wu, Abdullah Mueen, Maria A. Zuluaga, and Eamonn Keogh. Matrix Profile XXIV: Scaling Time Series Anomaly Detection to Trillions of Datapoints and Ultra-fast Arriving Data Streams. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '22, page 1173–1182, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3534678.3539271.
- [LZPK18] Michele Linardi, Yan Zhu, Themis Palpanas, and Eamonn Keogh. Matrix Profile X: VALMOD - Scalable Discovery of Variable-Length Motifs in Data Series. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1053–1066, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3183713.3183744.
- [MAA<sup>+</sup>21] Ryan Mercer, Sara Alaei, Alireza Abdoli, Shailendra Singh, Amy Murillo, and Eamonn Keogh. Matrix Profile XXIII: Contrast Profile: A Novel Time Series Primitive that Allows Real World Classification. In *2021 IEEE International Conference on Data Mining (ICDM)*, pages 1240–1245, 2021. doi:10.1109/ICDM51629.2021.00151.
- [Mai21] Matthias Maiterth. A reference model for integrated energy and power management of hpc systems. Ludwig-Maximilians-Universität München, September 2021. URL: <http://nbn-resolving.de/urn:nbn:de:bvb:19-286250>.
- [Max23] Max Plank Computing and Data Facility. *Supercomputer Raven at Max Plank Computing and Data Facility*, Online, Accessed on December 16, 2023. URL: <https://www.mpcdf.mpg.de/services/supercomputing/raven>.
- [McC95] John D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. pages 19–25, December 1995. URL: <http://dx.doi.org/10.26153/tsw/13794>.
- [ML14] Marius Muja and David G. Lowe. Scalable Nearest Neighbor Algorithms for High Dimensional Data. volume 36, pages 2227–2240, 2014. doi:10.1109/tpami.2014.2321376.
- [Moo06] Gordon E. Moore. Cramming More Components onto Integrated Circuits, Reprinted from *Electronics*, Volume 38, Number 8, April 19, 1965, pp.114 ff. volume 11, pages 33–35, 2006. doi:10.1109/N-SSC.2006.4785860.

- [MSR22] Burak Mete, Martin Schulz, and Martin Ruefenacht. Predicting the Optimizability for Workflow Decisions. In *2022 IEEE/ACM Third International Workshop on Quantum Computing Software (QCS)*, pages 68–74, 2022. doi:10.1109/QCS56647.2022.00013.
- [MZ15] Aleksander Movchan and Mikhail L. Zymbler. Time Series Subsequence Similarity Search Under Dynamic Time Warping Distance on the Intel Many-core Accelerators. In *SISAP*, 2015. doi:10.1007/978-3-319-25087-8\_28.
- [Nat19] Kousik Nath. An in-Depth Look at Database Indexing. 2019. Online, Accessed on December 16, 2023. URL: <https://www.freecodecamp.org/news/database-indexing-at-a-glance-bb50809d48bd/>.
- [NC23] NVIDIA-Corporation. Modern GPULibrary, Release: 2.0, (accessed June 19, 2023). URL: <https://github.com/moderngpu/moderngpu/wiki>.
- [Nes23] NestedSoftware. Incremental Average and Standard Deviation with Sliding Window, Online, Accessed on December 16, 2023. URL: <https://nestedsoftware.com/2019/09/26/incremental-average-and-standard-deviation-with-sliding-window-470k.176143.html>.
- [Net20] Alessio Netti. HPC-ODA dataset collection, September 2020. doi:10.5281/zenodo.3701440.
- [Net22] Alessio Netti. Holistic and Portable Operational Data Analytics on Production HPC Systems. page 210, 2022. URL: [https://mediatum.ub.tum.de/doc/1620116/smwwljju30ph0bz1ewhiw9tbeb.Netti\\_Alessio\\_PhD\\_Dissertation.pdf](https://mediatum.ub.tum.de/doc/1620116/smwwljju30ph0bz1ewhiw9tbeb.Netti_Alessio_PhD_Dissertation.pdf).
- [Nie20] Aileen Nielsen. Practical Time Series Analysis. O’Reilly Media, 2020. URL: <https://learning.oreilly.com/library/view/practical-time-series/9781492041641/>.
- [NMA<sup>+</sup>19] Alessio Netti, Micha Müller, Axel Auweter, Carla Guillen, Michael Ott, Daniele Tafani, and Martin Schulz. From Facility to Application Sensor Data: Modular, Continuous and Holistic Monitoring with DCDB, 2019. doi:10.1145/3295500.3356191.
- [NS07] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’07*, pages 89–100, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1250734.1250746.
- [NSO<sup>+</sup>21] Alessio Netti, Woong Shin, Michael Ott, Torsten Wilde, and Natalie Bates. A Conceptual Framework for HPC Operational Data Analytics. In *2021*

- IEEE International Conference on Cluster Computing (CLUSTER)*, pages 596–603, 2021. doi:10.1109/Cluster48925.2021.00086.
- [NVF20] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. CUDA, release: 10.2.89, 2020. URL: <https://developer.nvidia.com/cuda-toolkit>.
- [NVI23a] NVIDIA. Cooperative Groups API, (accessed June 19, 2023). URL: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cooperative-groups>.
- [NVI23b] NVIDIA. CUB Library, Release: 1.14.0, (accessed June 19, 2023). URL: <https://nvlabs.github.io/cub>.
- [NVI23c] NVIDIA. CUDA Math API, (accessed June 19, 2023). URL: <https://docs.nvidia.com/cuda/cuda-math-api/index.html>.
- [NVI23d] NVIDIA. Stream Management API, (accessed June 19, 2023). URL: [https://docs.nvidia.com/cuda/cuda-runtime-api/group\\_\\_CUDA\\_\\_STREAM.html#group\\_\\_CUDA\\_\\_STREAM](https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDA__STREAM.html#group__CUDA__STREAM).
- [Ope08] OpenMP Architecture Review Board. OpenMP Application Program Interface, May 2008. URL: <https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf>.
- [Pfe19] Gabriel Pfeilschifter. Time Series Analysis With Matrix Profile on HPC Systems. Master thesis, Technische Universität München, 2019. URL: <https://mediatum.ub.tum.de/doc/1471292/document.pdf>.
- [Pod] Exascale Computing Project Podcast. Getting Computing Luminary Jack Dongarra’s Perspective on the Exascale Computing Project. <https://www.exascaleproject.org/getting-computing-luminary-jack-dongarras-perspective-on-the-exascale-computing-project/>. Online, Accessed on Sep 12, 2023. URL: <https://www.exascaleproject.org/getting-computing-luminary-jack-dongarras-perspective-on-the-exascale-computing-project/>.
- [PSS<sup>+</sup>16] Md. Mostofa Ali Patwary, Nadathur Satish, Narayanan Sundaram, Jialin Liu, Peter J. Sadowski, Evan Racah, Surendra Byna, Craig Tull, Wahid Bhimji, Prabhat, and Pradeep Dubey. PANDA: Extreme Scale Parallel K-Nearest Neighbor on Distributed Architectures. pages 494–503, 2016. doi:10.1109/ipdps.2016.57.
- [RCM<sup>+</sup>12] Thanawin Rakthanmanon, Bilson J. L. Campana, Abdullah Al Mueen, Gustavo E. A. P. A. Batista, M. Brandon Westover, Qiang Zhu, Jesin Zakaria, and Eamonn J. Keogh. Searching and Mining Trillions of Time Series Subsequences under Dynamic Time Warping. volume 2012, pages 262 – 270, 2012. doi:10.1145/2339530.2339576.

- [Rei13] James R Reinders. AVX-512 Instructions. 2013. URL: <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-avx-512-instructions.html>.
- [Res22] Cambrian AI Research. The Data Center Architecture for Graphcore Computing, 2022. URL: <https://www.graphcore.ai/hubfs/Graphcore-Mk2-IPU-System-Architecture-GC.pdf>.
- [RHP16] D. Rouwenhorst, J. Hermann, and W. Polifke. Online Monitoring of Thermoacoustic Eigenmodes in Annular Combustion Systems Based on a State-Space Model. volume 139, page 021502, 09 2016. doi:10.1115/1.4034260.
- [RKS<sup>+</sup>23] Amir Raoofy, Roman Karlstetter, Martin Schreiber, Carsten Trinitis, and Martin Schulz. Overcoming Weak Scaling Challenges in Tree-Based Nearest Neighbor Time Series Mining. In *High Performance Computing: 38th International Conference, ISC High Performance 2023, Hamburg, Germany, May 21-25, 2023, Proceedings*, pages 317–338, Berlin, Heidelberg, 2023. Springer-Verlag. doi:10.1007/978-3-031-32041-5\_17.
- [RKY<sup>+</sup>20] Amir Raoofy, Roman Karlstetter, Dai Yang, Carsten Trinitis, and Martin Schulz. Time Series Mining at Petascale Performance. In Ponnuswamy Sadayappan, Bradford L. Chamberlain, Guido Juckeland, and Hatem Ltaief, editors, *High Performance Computing*, pages 104–123, Cham, 2020. Springer International Publishing.
- [Rok85] V Rokhlin. Rapid Solution of Integral Equations of Classical Potential Theory. volume 60, pages 187–207, 1985. doi:10.1016/0021-9991(85)90002-6.
- [RS02] John F. Roddick and Myra Spiliopoulou. A Survey of Temporal Knowledge Discovery Paradigms and Methods. volume 14, pages 750–767, 2002. doi:10.1109/TKDE.2002.1019212.
- [RS19] Parikshit Ram and Kaushik Sinha. Revisiting Kd-Tree for Nearest Neighbor Search. KDD '19, pages 1378–1388, New York, NY, USA, 2019. Association for Computing Machinery.
- [RTL<sup>+</sup>22] Martin Ruefenacht, BRUNO G Taketani, PASI Lähteenmäki, VILLE Bergholm, DIETER Kranzlmüller, LAURA Schulz, and MARTIN Schulz. Bringing Quantum Acceleration to Supercomputers, 2022. URL: [https://www.quantum.lrz.de/fileadmin/QIC/Downloads/IQM\\_HPC-QC-Integration-Whitepaper.pdf](https://www.quantum.lrz.de/fileadmin/QIC/Downloads/IQM_HPC-QC-Integration-Whitepaper.pdf).
- [Rus78] Richard M. Russell. The CRAY-1 Computer System. volume 21, pages 63–72, New York, NY, USA, jan 1978. Association for Computing Machinery. doi:10.1145/359327.359336.



- [RVR<sup>+</sup>20] Jose C Romero, Antonio Vilches, Andrés Rodríguez, Angeles Navarro, and Rafael Asenjo. Scrimpc: Scalable Matrix Profile On Commodity Heterogeneous Processors. *The Journal of Supercomputing*, pages 1–22, 2020. doi:10.1007/s11227-020-03199-w.
- [Sch87] Paul B. Schneck. *The CDC STAR-100*, pages 99–117. Springer US, Boston, MA, 1987. doi:10.1007/978-1-4615-7957-1\_5.
- [SDSM23] Erich Strohmaier, Jack Dongarra, Horst Simon, and Martin Meuer. TOP500. <https://www.top500.org>, Online, Accessed on August 16, 2023. URL: <https://top500.org/>.
- [SGM<sup>+</sup>08] Martin Schulz, Jim Galarowicz, Don Maghrak, William Hachfeld, David Montoya, and Scott Cranford. Open — SpeedShop: An Open Source Infrastructure for Parallel Performance Analysis. volume 16, pages 105–121, NLD, apr 2008. IOS Press. doi:10.1155/2008/713705.
- [SHWG15] Benedikt Steinbusch, M. Henkel, M. Winkel, and P. Gibbon. A Massively Parallel Barnes-Hut Tree Code with Dual Tree Traversal. In *PARCO*, 2015. URL: <https://juser.fz-juelich.de/record/885124/files/ParCo2015-paper.pdf>.
- [SKJJ16] Rushikesh Salunkhe, Aniket D Kadam, Naveenkumar Jayakumar, and Shashank Joshi. Luster A Scalable Architecture File System: A Research Implementation On Active Storage Array Framework With Luster File System. In *2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT)*, pages 1073–1081, 2016. doi:10.1109/ICEEOT.2016.7754852.
- [SMM06] Sameer Shende, Allen Malony, and Alan Morris. Workload Characterization Using the TAU Performance System. volume 4699, pages 289–296, 06 2006. doi:10.1007/978-3-540-75755-9\_35.
- [SMN<sup>+</sup>10] D. Sart, A. Mueen, W. Najjar, E. Keogh, and V. Niennattrakul. Accelerating Dynamic Time Warping Subsequence Search with GPUs and FPGAs. In *2010 IEEE International Conference on Data Mining*, pages 1001–1006, Dec 2010. doi:10.1109/icdm.2010.21.
- [SSFZ<sup>+</sup>18] Nader S. Shakibay Senobari, Gareth Funning, Zachary Zimmerman, Yan Zhu, and Eamonn J. Keogh. Using the Similarity Matrix Profile to Investigate Foreshock Behavior of the 2004 Parkfield Earthquake. *AGUFM*, 2018:S51B–03, 2018. URL: <https://www.cs.ucr.edu/~eamonn/MatrixProfile2004Parkfieldearthquake.pdf>.
- [Sut13] Herb Sutter. The Free Lunch Is Over A Fundamental Turn Toward Concurrency in Software. 2013. URL: <https://api.semanticscholar.org/CorpusID:264732939>.



- [SYBK16] Diego Furtado Silva, Chin-Chia Michael Yeh, Gustavo E. A. P. A. Batista, and Eamonn J. Keogh. Simple: Assessing music similarity using subsequences joins. In *International Society for Music Information Retrieval Conference*, 2016. URL: <https://api.semanticscholar.org/CorpusID:14118722>.
- [SYK<sup>+</sup>19] Jie Shi, Nanpeng Yu, Eamonn Keogh, Heng Kevin Chen, and Koji Yamashita. Discovering and Labeling Power System Events in Synchrophasor Data with Matrix Profile. In *2019 IEEE Sustainable Power and Energy Conference (iSPEC)*, pages 1827–1832. IEEE, 2019. doi:10.1109/ispec48194.2019.8975286.
- [SZSF<sup>+</sup>19] Zachary Schall-Zimmerman, Nader Shakibay Senobari, Gareth J. Funning, Evangelos E. Papalexakis, Samet Oymak, Philip Brisk, and Eamonn J. Keogh. Matrix Profile XVIII: Time Series Mining in the Face of Fast Moving Streams using a Learned Approximate Matrix Profile. pages 936–945, 2019. doi:10.1109/icdm.2019.00104.
- [Tay] Stewart Taylor. *Optimizing Applications for Multi-core Processors, Using the Intel Integrated Performance Primitives*. Intel Press.
- [Tha20] Alaa Tharwat. Classification assessment methods. 2020. doi:10.1016/j.aci.2018.08.003.
- [THW10] J. Treibig, G. Hager, and G. Wellein. LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*, San Diego CA, 2010. doi:10.1109/icppw.2010.38.
- [TJYD09] Daniel Terpstra, Heike Jagode, Haihang You, and Jack J. Dongarra. Collecting Performance Data with PAPI-C. In Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2009 - Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing, September 2009, ZIH, Dresden*, pages 157–173. Springer, 2009. doi:10.1007/978-3-642-11261-4\_11.
- [TKB20] Markus Thill, Wolfgang Konen, and Thomas Bäck. MarkusThill/M-GAB: The Mackey-Glass Anomaly Benchmark, April 2020. doi:10.5281/zenodo.3760086.
- [TMB20] Giuseppe Tagliavini, Andrea Marongiu, and Luca Benini. FlexFloat: A Software Library for Transprecision Computing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39:145–156, 2020. doi:10.1109/TCAD.2018.2883902.

- [TS] Chair of Computer Architecture TUM and Parallel Systems. Gas Turbine Optimization using Big Data (Turbo), research project funded by Funded by Bavarian Research Foundation. Online, Accessed on June 23, 2023. URL: <https://turbo.caps.in.tum.de/>.
- [Tur50] A. M. Turing. Computing Machinery and Intelligence. volume 59, pages 433–460. [Oxford University Press, Mind Association], 1950. URL: <http://www.jstor.org/stable/2251299>.
- [VC06] Jeffery Vetter and Christopher Ch�mbreau. A Light-Weight MPI Profiler. In *Lawrence Livermore National Laboratory*, 2006. URL: <https://github.com/LLNL/mpiP>.
- [vdGG11] Robert van de Geijn and Kazushige Goto. *BLAS (Basic Linear Algebra Subprograms)*, pages 157–164. Springer US, Boston, MA, 2011. doi:10.1007/978-0-387-09766-4\_84.
- [VDM14] Laurens Van Der Maaten. Accelerating t-SNE Using Tree-Based Algorithms. volume 15, pages 3221–3245. JMLR.org, January 2014. URL: <http://jmlr.org/papers/v15/vandermaaten14a.html>.
- [VdSB<sup>+</sup>18] Sudharshan S. Vazhkudai, Bronis R. de Supinski, Arthur S. Bland, Al Geist, James Sexton, Jim Kahle, Christopher J. Zimmer, Scott Atchley, Sarp Oral, Don E. Maxwell, Veronica G. Vergara Larrea, Adam Bertsch, Robin Goldstone, Wayne Joubert, Chris Ch�mbreau, David Appelhans, Robert Blackmore, Ben Casses, George Chochia, Gene Davison, Matthew A. Ezell, Tom Gooding, Elsa Gonsiorowski, Leopold Grinberg, Bill Hanson, Bill Hartner, Ian Karlin, Matthew L. Leininger, Dustin Leverman, Chris Marroquin, Adam Moody, Martin Ohmacht, Ramesh Pankajakshan, Fernando Pizzano, James H. Rogers, Bryan Rosenburg, Drew Schmidt, Mallikarjun Shankar, Feiyi Wang, Py Watson, Bob Walkup, Lance D. Weems, and Junqi Yin. The design, deployment, and evaluation of the coral pre-exascale systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18. IEEE Press, 2018. doi:10.1109/sc.2018.00055.
- [vN45] John von Neumann. First Draft of a Report on the EDVAC. Technical report, 1945. URL: <https://web.mit.edu/STS.035/www/PDFs/edvac.pdf>.
- [WD96] David W Walker and Jack J Dongarra. MPI: A Standard Message Passing Interface. volume 12, pages 56–68. ASFRA BV, 1996. URL: <https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf>.
- [WMD<sup>+</sup>13] Xiaoyue Wang, Abdullah Mueen, Hui Ding, Goce Trajcevski, Peter Scheuermann, and Eamonn Keogh. Experimental Comparison of Representation Methods and Distance Measures for Time Series Data. vol-

- ume 26, pages 275–309, USA, mar 2013. Kluwer Academic Publishers. doi:10.1007/s10618-012-0250-5.
- [WWP09] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. volume 52, pages 65–76, New York, NY, USA, apr 2009. Association for Computing Machinery. doi:10.1145/1498765.1498785.
- [XB16] Bo Xiao and George Biros. Parallel Algorithms for Nearest Neighbor Search Problems in High Dimensions. volume 38, pages S667–S699, 2016. doi:10.1137/15M1026377.
- [Yan20] Dai Yang. Fault Tolerant Optimizations for High Performance Computing Systems. 2020. URL: <https://mediatum.ub.tum.de/doc/1518787/1518787.pdf>.
- [YFS21] L. Yang, Alyson Fox, and G. Sanders. Rounding Error Analysis of Mixed Precision Block Householder QR Algorithms. *SIAM J. Sci. Comput.*, 43:A1723–A1753, 2021. doi:10.1137/19m1296367.
- [YHA<sup>+</sup>15] Chenhan D. Yu, Jianyu Huang, Woody Austin, Bo Xiao, and George Biros. Performance Optimization for the k-Nearest Neighbors Kernel on x86 Architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '15*, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2807591.2807601.
- [YHK16] C. M. Yeh, H. V. Herle, and E. Keogh. Matrix Profile III: The Matrix Profile Allows Visualization of Salient Subsequences in Massive Time Series. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 579–588, Dec 2016. doi:10.1109/ICDM.2016.0069.
- [YJG03] Andy B. Yoo, Morris A. Jette, and Mark Grondona. SLURM: Simple Linux Utility for Resource Management. In Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 44–60, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.
- [YKK17] C. M. Yeh, N. Kavantzias, and E. Keogh. Matrix Profile VI: Meaningful Multidimensional Motif Discovery. In *2017 IEEE International Conference on Data Mining (ICDM)*, pages 565–574, Nov 2017. doi:10.1109/ICDM.2017.66.
- [YZU<sup>+</sup>16] C. M. Yeh, Y. Zhu, L. Ulanova, N. Begum, Y. Ding, H. A. Dau, D. F. Silva, A. Mueen, and E. Keogh. Matrix Profile I: All Pairs Similarity Joins Time Series: A Unifying View That Includes Motifs, Discords and Shapelets. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 1317–1322, Dec 2016. doi:10.1109/ICDM.2016.0179.

- [YZU<sup>+</sup>18] Chin-Chia M. Yeh, Yan Zhu, Liudmila Ulanova, Nurjahan Begum, Yifei Ding, Hoang A. Dau, Zachary Zimmerman, Diego F. Silva, Abdullah Mueen, and Eamonn Keogh. Time Series Joins, Motifs, Discords And Shapelets: A Unifying View That Exploits The Matrix Profile. *Data Mining and Knowledge Discovery*, 32(1):83–123, 2018. doi:[10.1007/s10618-017-0519-9](https://doi.org/10.1007/s10618-017-0519-9).
- [ZGS<sup>+</sup>20] Yan Zhu, Shaghayegh Gharghabi, Diego Furtado Silva, Hoang Anh Dau, Chin-Chia Michael Yeh, Nader Shakibay Senobari, Abdulaziz Almaslukh, Kaveh Kamgar, Zachary Zimmerman, Gareth Funning, Abdullah Mueen, and Eamonn Keogh. The Swiss Army Knife of Time Series Data Mining: Ten Useful Things You Can Do With the Matrix Profile And Ten Lines of Code. *Data Mining and Knowledge Discovery*, 34(4):949–979, 2020. doi:[10.1007/s10618-019-00668-6](https://doi.org/10.1007/s10618-019-00668-6).
- [ZINK17] Yan Zhu, Makoto Imamura, Daniel Nikovski, and Eamonn Keogh. Matrix Profile VII: Time Series Chains: A New Primitive for Time Series Data Mining. In *2017 IEEE International Conference on Data Mining (ICDM)*, pages 695–704, 2017. doi:[10.1109/ICDM.2017.79](https://doi.org/10.1109/ICDM.2017.79).
- [ZKS<sup>+</sup>19] Zachary Zimmerman, Kaveh Kamgar, Nader Senobari, Brian Crites, Gareth Funning, Philip Brisk, and Eamonn Keogh. Matrix Profile XIV: Scaling Time Series Motif Discovery with GPUs to Break a Quintillion Pairwise Comparisons a Day and Beyond. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 74–86, 2019. doi:[10.1145/3357223.3362721](https://doi.org/10.1145/3357223.3362721).
- [ZXT<sup>+</sup>21] Xiangtian Zheng, Nan Xu, Loc Trinh, Dongqi Wu, Tong Huang, S. Sivarajani, Yan Liu, and Le Xie. PSML: A Multi-scale Time-series Dataset for Machine Learning in Decarbonized Energy Grids (Dataset), August 2021. doi:[10.5281/zenodo.5130612](https://doi.org/10.5281/zenodo.5130612).
- [ZYZ<sup>+</sup>18] Y. Zhu, C. M. Yeh, Z. Zimmerman, K. Kamgar, and E. Keogh. Matrix Profile XI: SCRIMP++: Time Series Motif Discovery at Interactive Speeds. In *2018 IEEE International Conference on Data Mining (ICDM)*, pages 837–846, Nov 2018. doi:[10.1109/ICDM.2018.00099](https://doi.org/10.1109/ICDM.2018.00099).
- [ZZS<sup>+</sup>18] Yan Zhu, Zachary Zimmerman, Nader Shakibay Senobari, Chin-Chia Michael Yeh, Gareth Funning, Abdullah Mueen, Philip Brisk, and Eamonn Keogh. Matrix Profile II: Exploiting a Novel Algorithm and GPUs to Break the One Hundred Million Barrier for Time Series Motifs and Joins. volume 54, pages 203–236. Springer, 2018. doi:[10.1109/ICDM.2016.0085](https://doi.org/10.1109/ICDM.2016.0085).

Left blank intentionally.