Technical Univertsity of Munich

Department of Informatics

Master’s thesis Robotics, Cognition, Intelligence

Time Series Analysis with Matrix Profile on HPC Systems

Gabriel Pfeilschifter

Technical Univertsity of Munich

Department of Informatics

Master’s thesis Robotics, Cognition, Intelligence

Time Series Analysis with Matrix Profile on HPC Systems

Zeitreihenanalyse mittels Matrix Profile auf HPC Systemen

Supervisor	Univ.-Prof. Dr. rer. nat. Martin Schulz Chair of Computer Architecture and Parallel Systems
Advisors	M.Sc. Amir Raoofy M.Sc. Roman Karlstetter Chair of Computer Architecture and Parallel Systems
Author	Gabriel Pfeilschifter
Submission	München, den 15.01.2019

Declaration of Authorship

I confirm that this master's thesis is my own work and I have documented all sources and material used.

München, 15.01.2019 Gabriel Pfeilschifter

Abstract

Time series data is naturally formed by repeated measurements over time and their analysis is in various science and engineering disciplines precedes the digital age. With progressing digitalization the amount and detail of such data is ever more growing and drives the demand for performant data mining tools to gain insights from the increasing amounts of data. The matrix profile proposed is a data structure which can serve in a variety of time series data mining tasks like motif search or clustering. Such analysis can be performed with little effort given the matrix profile, but obtaining the latter is is a computationally intensive task.

In this work we investigate migration of the matrix profile computation to a high performance computing cluster utilizing the MPI standard. Highly optimized parallel routines of MPI and the vast number of available computing elements on such clusters allow users to scale their algorithms in terms of runtime and level of detail according to their needs within the most distant limits.

We present an approach, which enables the analysis of previously intractably large time series based on the matrix profile. It provides nearly unlimited scalability to solve large problems within reasonable runtimes according to the users needs.

In this work we provide a brief overview of time series analysis in general and review the contributions of the matrix profile in the field. We present applicable sequential optimizations to the SCRIMP kernel, on which we base our implementation. In particular we examine two parallelization approaches: the first one is based on suggestions from literature. In contrast to it, the second parallelization approach proposed by us in this work provides scalability to longer time series, as it is not bound by single-node hardware. To asses the quality of the implementations we perform a series of scaling experiments. We investigate potential scaling bottlenecks of the implementations and fit a runtime model to predict tractability of time series analysis when applying the approach in practice.

Acknowledgements

I want to thank Gerd and Dr. Lenz for motivating me to pursue this work. Special gratitude goes to Hildegard, who always had an open ear and kept me going in times of struggle.

Many thanks to Amir and Roman for their invaluably wise advice, their patience and in particular for the spontaneous and fruitful discussions.

Finally my gratitude goes to Prof. Schulz and all the other persons at the CAPS chair for enabling me to write this work and for the welcoming and supportive atmosphere.

Contents

1 Introduction ... 3
2 Related Work ... 5
2.1 Time Series Analysis ... 5
2.2 Matrix Profile ... 8
2.2.1 Applications in Time Series Analysis ... 8
2.2.2 Algorithms ... 12
2.3 HPC Approaches to Time Series Analysis ... 13
3 Background ... 16
3.1 Definitions ... 16
3.1.1 The Matrix Profile ... 16
3.1.2 Z-Normalized Euclidean Distance ... 19
3.2 Algorithms ... 20
3.2.1 SCRIMP ... 21
3.3 Performance of Parallel Programs ... 22
3.3.1 Kernel Performance ... 24
3.3.2 Scalability ... 26
4 Sequential Optimization ... 30
4.1 Arithmetic Kernel ... 30
4.2 Algorithmic Kernel ... 32
4.2.1 STOMP/SCRIMP ... 32
4.2.2 Blocking Iteration Schemes ... 34
4.3 Vectorization ... 39
4.4 Comparison to SCAMP ... 40
5 Parallelization ... 42
5.1 Trivial Parallelization ... 42
5.1.1 Algorithm ... 42
5.1.2 Mapping to the MPI ... 47
5.1.3 Theoretical Performance Analysis ... 49
5.2 Distributed parallelization ... 51
5.2.1 Algorithm ... 51
5.2.2 Mapping to the MPI ... 62
5.2.3 Theoretical Performance Analysis ... 63
5.2.4 Comparison to SCAMP ... 66
6 Experiments ... 68
6.1 Sequential Optimization ... 68
6.1.1 Comparison of Kernels ... 68
6.1.2 Blocking Kernel Performance ... 73
6.2 Trivial Parallelization ... 75
6.3 Distributed Parallelization ... 81
6.3.1 Performance Model ... 81

6.3.2 Parallelization Bottlenecks 87
6.4 Comparison of Implementations 94
6.4.1 Trivial and Distributed Parallelization 94
6.4.2 Comparison to SCAMP 96
7 Discussion 100
7.1 MPI Implementation Choices 100
7.2 Scalability of the Approach 102
7.3 Performance Model 103
7.4 Comparison to SCAMP 104
7.5 Summary and Outlook 105
8 Appendix 118
8.1 Experimental Setup 118
8.2 Numerical Results 121
8.3 Textual Listings 128
8.4 Images 131

1. Introduction

We call the current time period close to the beginning of the 21 century the *information age*, as major changes in society are driven by technological advances in communication and information processing. The major resource of information are digital data. They are stated as equally valuable as raw materials by leading business men[1], terms like *big data* and *data mining* are circulating in popular media.

Despite the public attention, scientific insights and economical value had been generated based on data for centuries. In particular, repeated recordings of the same measurement quantities, called time series data, enabled understanding of dynamical processes, such as the movement of stars or market behavior. The drastic changes which cause the prominence of data in recent years are the insights gained from advancing digital processing techniques and the ever growing amount of data collected from growing numbers of network connected sensors. Inherently, data generated by digital sensors form time series and the growing amount of collected data motivates advances in the field of time series data mining.

While increasing amounts and quality of available data promise new insights, such are limited by the available analysis tools and their applicability to the large data sets. Yeh *et al.* [1] propose a analysis tool, called the *matrix profile*, which enables a variety of time series analysis which are useful for example in monitoring of industrial processes, fault-detection, root-cause analysis or the exploration of unknown data sets. In this work we focus on the task to extend the efficiency and scalability of the matrix profile computation in order to provide an efficient and scalable analysis tool to meet the demand for processing of ever more detailed or lengthy time series data. Specifically we target a high performance computing cluster, as it promises the highest scalability available for scientific investigations. The remaining parts of our thesis are structured according to following outline:

In chapter 2 we start with an overview on time series analysis to locate the matrix profile and its contributions within the field. We give a overview of the available data analysis with the technique as well as on implementations presented in the literature and conclude our review of related work with presentation of different HPC efforts in the time series analysis domain. Chapter 3 provides the necessary theoretical details on the algorithm from the literature for understanding and discussion of our work. It further reviews techniques and metrics which we will use for performance measurement and discussion. Chapters 4 to 5 explain our approaches and implementation choices for optimization and parallelization. A set of experiments is reported and analyzed in chapter 6 to examine their effectiveness. In particular the implementations scaling behavior and potential bottlenecks are examined. Finally in chapter 7 we discuss choices and experiences of with work. The results and contributions are criti-

[1] Siemens CEO Joe Kaeser https://economictimes.indiatimes.com/magazines/panache/data-is-the-21st-centurys-oil-says-siemens-ceo-joe-kaeser/articleshow/64298125.cms

cally reviewed and compared to a similar approach of Zimmerman *et al.* [2], which became available during our work.

2. Related Work

We start our work with a general review of the field of time series analysis in section 2.1 to provide an overview for locating the matrix profile within the field. It starts with general terms and traditional statistical techniques before turning towards data-mining approaches. With that background we outline the application of the matrix profile in such analysis in section 2.2 and give an overview of the algorithms found in literature. We conclude our review of related work in section 2.3 with previous efforts for high performance time series analysis to highlight the motivation for our approach.

2.1 Time Series Analysis

Subsequent measurement values acquired by electrical sensors naturally form time series and are an exhaustive source of such data. As also manual recordings are often acquired over time, interest in their analysis predates the digital age, e.g. in economics and natural sciences. Due to their simplified description and todays preferred digital processing technology, typically discrete time series with a finite number of records are considered. As Chatfield [3] explains, the values can be immediate samples of a variable or aggregates over the sampling time interval, i.e represent densities. If several recorded variables are dealt with, the time series is called multivariate. In case that the values form an exactly predictable process, the series is named deterministic, otherwise stochastic.

As such data are recorded in manifold domains, application fields span a broad range, including a wide span of sciences. For example in bio-medicine [4] epidemiological data as well as patient data like electrocardiograms are considered. Further examples are data collected from a gamified behavioral experiment in psychology [5], seismic amplitudes in geology [6], inflation rates [7] or electrical prices [8] in economy, star light amplitudes in physics [9], the amount of weekly property crime in criminology [10] or network traffic in computer science[11].Furthermore there are technological applications like retrieval of music from humming [12] or monitoring of semiconductor manufacturing processes [13].

One can consider time series analysis from two partially overlapping perspectives. It forms a sub-discipline of statistics from a mathematical point of view and on the other hand it is a variant of data mining focused on a distinct form of data. We will give a short overview of the concerns of both fields.

Statistical Discipline

Time series analysis forms a branch of statistical science. Chatfield [3] gives an introduction to this field and links to more in depth literature. He distinguishes four variants of analysis on

time series data depending on the objective. Namely description, explanation, prediction and control, whose meaning will be briefly discussed in the next paragraph. A further distinction can be made based on the type of analyzed data: Analysis may be performed on the raw time series values, or on transforms of it, typically spectral ones like the Fourier transform. This raises the categorization into time-domain and spectral analysis.

Let us briefly review the distinct analysis objectives named by Chatfield [3]. *Descriptive analysis* aims at discovering behavior of the data, for example oscillations or long term trends. It can be as simple as plotting the data over time and is always involved in any analysis, as it is necessary for deciding upon specific models or techniques to apply in further steps. Typical tasks are description of trends and periodicity in the data and possibly removing them, e.g. by differencing and regression. A very common tool is also plotting of the auto-covariance function to detect properties of the time series and reason about suitable models. For some tasks though, descriptive analysis is sufficient on itself.

Understanding the mechanism, which is responsible for generation of the time series is the objective of *explanatory analysis*. This involves fitting of adequate models to the data. Important models are for example auto-regressive processes, which model an internal state as being dependent on a fixed range of preceding states, moving average models, which model the series as a weighted sum of another stochastic series or the ARIMA (auto regressive integrated moving average) model. These are a mixture of auto-regressive and moving-average models and in contrast to the previous ones can also cope with non-stationary processes (e.g. time series with a long term trend). The task of *prediction*, also named forecasting, is concerned with the extrapolation of the series beyond the last recorded value. The possibility to do so is inherently limited by the complexity of the underlying process and accuracy decreases with the desired lead time into the future. Simple point forecasts are only concerned with predicting the "exact" value of the time series at a desired future time point.

More extensive approaches produce prediction intervals or even probability distributions, which is called density forecasting. Often models from explanatory analysis are exploited, for example the popular Box-Jenkins approach for univariate time series builds on the ARIMA model. Sometimes also subjective input from the analyst is considered.

Lastly *control* is concerned with manipulation of the future progress of the time series. Its most prominent variant is control theory in engineering sciences. While subsuming it as a form of time series analysis Chatfield [3, pp. 266 sq.] also points out, that research and methodology in control theory exhibits quite some independence.

Data Mining Variant

Considering time series analysis as a data-mining variant, one treats, in the words of Kleist [14, p. 1], with analytic processes of knowledge discovery in large and complex data sets.

Various common tasks of the field also apply to time series, notably clustering, knowledge discovery, classification, rule discovery and again prediction. Regarding time series, for all these tasks efforts have been made to adopt classical data mining approaches but also some highly specific solutions had been proposed. As also tools from statistical time series analysis had been integrated, the two perspectives are overlapping and complementary. Further there are typical preprocessing steps specific to time series data, which are often shared by the different fields. For example different data representations, similarity measures, indexing or segmentation techniques.

Within data mining, one can subsume time series analysis in the more general field of temporal data mining. The field also deals more generally with temporal data, e.g. irregularly recorded, timestamped objects like telecommunication signals and interested readers are referred to the survey of Roddick *et al.* [15] as a starting point. Let us briefly summarize the essence of Kleist's [14] review on time series data mining for each of the five stated tasks, before starting our examination of the matrix profile.

The task of *knowledge discovery* is dominated by pattern mining, which is concerned with finding patterns with distinct properties within the time series database. Typical targets are motifs, novelties and outliers. While motifs are patterns which can be found frequently, outliers are patterns which are rather unique and considered to be caused by a very different mechanism than the remaining time series. *Novelties*, or *surprising patterns*, on the other hand are data points, whose frequency is somehow defying the expectations. As this vague formulation suggests, a variety of different exact definitions of surprising patterns exist [16]. Algorithms for such tasks often are parameterized by a fixed pattern length. For example techniques from string processing had been employed for this task by low dimensional discretization of the time series, or machine learning techniques like neural networks. Further knowledge discovery tasks e.g. treat with detection of local periodic structures or repeating trends.

Clustering tries to group data according to a suitable similarity- or conversely distance metric: groups are formed such that the similarity of the members within one group is maximal while the inter-group similarity is minimized. Input of the clustering can be a set of time series from different sources or the set of sub-series of a single large data source. Typical approaches from machine learning are adopted for the task, e.g. hierarchical clustering creates a tree-structure from the bottom up by merging most similar pairs according to a similarity metric. Key challenges raised by time series are the high dimensionality, i.e the number of samples within time series. Further challenges are meaningful similarity metrics and the potentially high amount of outliers and noise, requiring pre-selection of the data.

Classification approaches try to create a mapping from time series to a given set of labels. The mapping is generated from a set of time series, which is accompanied by another series of label annotations. The mapping is then used at the so called inferencing time to retrieve

a labels for new data samples. Many famous machine learning approaches can be applied to time series data as well, like k-nearest neighbor classifiers and support vector machines. More tailored towards time series are Hidden Markov Models, which inherently model temporal evolution of a probabilistic state.

Rule discovery tries to reveal relations between variables. A typically considered set of rules are temporal association rules. A natural language example of such a rule, given by Roddick *et al.* [15], is that "chips and hot chili sauce are purchased together during winter". Very popular in this domain are again machine learning tools, e.g genetic algorithms or decision trees.

Finally prediction treats again with the extrapolation of time series data. Kleist [14] states the previously named statistical approaches as being dominant. Other applied techniques are for example neural networks or Hidden Markov Models.

2.2 Matrix Profile

The matrix profile, which our work is focused on, is a versatile tool in time-series data mining. It can aid in clustering, classification and pattern mining, where the last one is the most straight forward application. While it is not truly concerned with statistics, one could sort it into the the category of descriptive analysis in the terminology of [3] (as previously presented in this section).

The matrix profile names a data structure first introduced in [1] which is primarily describing the similarity of so called subsequences: cohesive subsets of the value series. The matrix profile is concerned with subsequences of a fixed length of samples.

Briefly summarized, the matrix profile and the accompanying profile index store the result of a all-pairs similarity join of time series subsequences: Given two time series A and B, for each subsequence of A the matrix profile stores an index and similarity measure of its most similar subsequence in time series B. This general approach is e.g. useful for analyzing data from different sources and denoted as subsequence *similarity-search* or AB-join. Particularly interesting in data mining is the examination of properties of a unknown data set acquired as a single time series A. This special case is called a AA- or self(-similarity)-join.

Based on the result of the self-similarity-join in form of the matrix profile, various data analysis are possible with low computational effort,. We give a brief overview in the next subsection. Afterwards a brief overview of known algorithms for its computation is given.

2.2.1 Applications in Time Series Analysis

One property, which makes the matrix profile particularly interesting is that various of the data mining tasks presented in section 2.1 can be performed with little effort, once it is given.

All of the analysis presented in the following paragraphs, except for the clustering, can be performed in a single pass over the the matrix profile, with linear time and space complexity $\mathcal{O}(n)$, where n denotes the length of the matrix profile (which is upper bound by and typically close to the input series length).

Motif and Discord Discovery

Motifs and discords are defined as special subsequences. Motifs are briefly described as subsequences which appear multiple times within the time series with a similar shape. Several different motifs can exist within a series. Figure 1 shows an example from the review of Torkamani *et al.* [17] on time-series motif discovery. Based on some distance measure, motif candidates are selected as subsequences with the low distances to each other and ranked. Ranking criteria are either the number of occurrences or the minimum distance between pairs of motif instances. The motifs are called *top-frequent* motifs in the first case and *range motifs* in the second one. In all cases so called *trivial matches* are not taken into consideration. Close-by subsequences, for example subsequences starting at time points i and $i+1$, have typically quite low distances but naturally do not form a motif [18].

Interested readers can find an overview on motif discovery in [19] and a composition of various algorithms in Torkamani *et al.* [17].

Motif search based on the matrix profile, as outlined in [20] (technically detailed in [1] and [21]) can provide motifs based on both ranking criteria. It further provides the specific benefit of interactivity compared to other methods: As the matrix profile contains information of a full all-pairs-similarity join, undesired motifs can be discarded and the next motifs obtained without re-running the motif computation on the input time series: in the analogy of texts, the most repeated words are articles like "the". Similar *stop-words* like calibration signals can occur in time series as motifs to be discarded. Furthermore applying weights with a *annotation vector* to the matrix profile enables focusing on motifs occurring close to known events, which can be helpful in root-cause analysis. Another advantage to rival methods highlighted by Yeh *et al.* [1] is that the matrix profile computes the full join and therefore no false negative can occur. The analysis result is exact, in contrast to many previous methods which prune computations by heuristics,

An interesting extension is also the applicability of the approach to multidimensional series, as presented in [22]. In contrast to other methods, the approach allows extraction of motifs spanning across a variable number of dimensions, including automated choice of this number. It is achieved by merging the information from matrix profiles, which are obtained for each track of the series independently.

Dual to motifs, *discords* are the subsequences with the highest distance to their nearest, non-trivial match [23]. Such sequences with the most unusual behavior within the time series

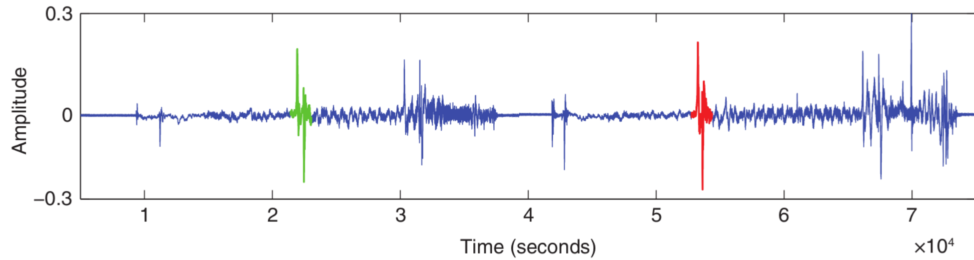
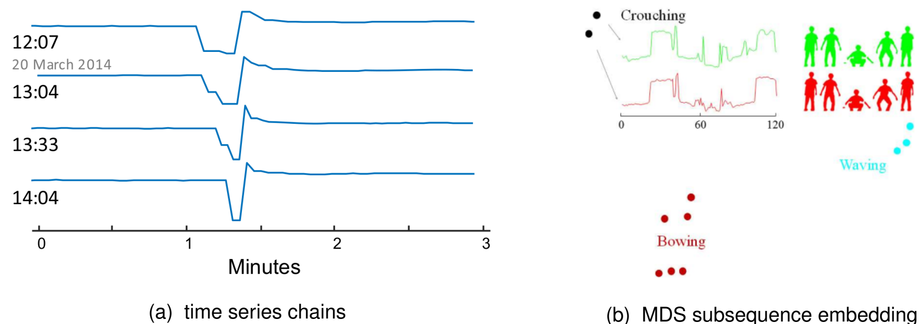


Figure 1 Example of a motif in a time series, by Torkamani *et al.* [17]: Highlighted in red and green are two instances of a motif, i.e. a repeated subsequence, in the real valued time-series.

can be used for the detection of subsequence anomalies, for which Varun *et al.* [24] compare different methods. Applications are for example failure analysis, predictive maintenance or medical diagnosis, a set of case studies is presented in [23]. As Keogh *et al.* [23] point out, the task does not allow application of divide-and-conquer approaches for optimization of the computations. Search for a arbitrary number of top discords of a given length is trivial given the matrix profile of the time series, as it contains the nearest neighbor distances of all subsequences.

Time series chains

In addition to motifs, the matrix profile allows exploration of time series chains [25], [26]. While a motif is composed of a group of instances with a limited distance to each other, time series chains are series of consecutively similar subsequences, as illustrated in figure 2a: while the distance between consecutive subsequences is low, distances between arbitrary pairs within the chain are allowed to grow up to infinity. In contrast to motifs, such chains enable exploration of evolving behavior, for example drifts in machine sensor data due to aging or changing environment conditions.



(a) time series chains

(b) MDS subsequence embedding

Figure 2 Subsequence clustering and time series chain examples: (a) by Zhu *et al.* [26] shows an example of a time series chain. The chain was extracted from an electrical power data set of a freezer. Note, how the very first and last subsequence are highly different in a sample-by-sample comparison while consecutive pairs are similar to each other, as the chain pattern is evolving over time. (b) shows the low-dimensional MDS embedding of a the case study by [27]. The algorithm successfully extracted a subset of meaningful repeated motion patterns (describing visual marker movements), which are semantically grouped together in a low-dimensional embedding.

Exploration of time series chains has been introduced first by Zhu *et al.* [25] based on the

matrix profile and to the best of our knowledge no alternative approach exists so far. The proposed method allows extraction of *anchored* chains, whose first element is a user specified anchor. Furthermore construction of the unique set of all non-overlapping unanchored chains allows to automatically extract for example the longest chain in the time series.

It is important to point out, that Zhu *et al.* [25] proposed the usage of two slightly modified matrix profiles: they capture for each subsequence the nearest neighbor among the sets of all preceding and following subsequences within the time series respectively. This is necessary to capture temporal relationships and requires a minor modification to the algorithm, which we do not consider in this work.

Clustering

The matrix profile is useful for *subsequence clustering*, as opposed to *whole clustering* of complete selected sequences in a database. As demonstrated by Keogh *et al.* [28], clustering of all subsequences within a time-series is condemned to fail and meaningful clustering requires selection of a reasonable candidate subset. Based on the matrix profile, Yeh *et al.* [27] present a method to perform the selection. The subsequent application of a multi-dimensional scaling (MDS) creates low-dimensional representations of the subsequences for visualization (see figure 2b) and can also serve to reduce the dimensionality for application of common whole-sequence clustering techniques. Namely the proposed scheme selects subsequences which are deemed similar to at least another one within the series. The selected set is one that achieves the maximum compressability, which is similar to the MDS objective. The approach in particular relies on the result of the full self-similarity join, as provided by the matrix profile, and is specifically appealing as it relies on the subsequence length of interest as the sole parameter.

While utility of the approach is demonstrated by several case studies, it is pointed out that it discards unique subsequences. E.g. inclusion of discords could be an important extension for specific application scenarios, which is straight-forward when the matrix profile is given.

Segmentation and classification

Yeh *et al.* [1] and Yeh *et al.* [21] also present a method, to segment a time-series, this is to divide it into regions of coherent behavior. Examples are idle or high load states in sensor data of industrial machines. The method is applicable, if each semantic region of the series contains specific characteristic motifs, which are never or only rarely occurring within different regions.

Yeh *et al.* [29] builds a classificator for time-series from a weakly labeled data set. The matrix profile is computed for a time-series which is the concatenation of all positively labeled

sequences of a class. Based on it, a set of of shape features, i.e. subsequences similar to the idea of motifs, and associated thresholds are selected such that optimum classification on the training data is achieved. Classification is performed by subsequence searches for the selected feature sequences within unknown data, for example with the MASS algorithm [30]. If similarity to a feature exceeds the computed threshold, subsequences are labeled positive. Advertised advantages compared to other clustering approaches are for example the robustness to labels with very inaccurate alignment, noise or false negative labels. Further the classification is not forced to assign one of the trained classes to any unknown or novel behavior

As the matrix profile is applied only to the manually labeled time series for construction of the classificator, this application is of minor interest for our work, as typically few labeled data is available due to the required manual work. Due to the limited data amount extreme scaling of the computations is typically not necessary. Also the segmentation technique is useful, but on itself might not justify the computational effort to compute matrix profiles for very large time-series.

2.2.2 Algorithms

As previously presented, the matrix profile is concerned with similarity search for fixed-length subsequences. This search-window length, which we denote as m, is the only input parameter required for the matrix profile computation. A series of consecutively improved algorithms for computation of the matrix profile had been proposed. While in the literature they are typically discussed for self-joins on a single time-series, all of them can be generalized and applied to AB joins.

The first algorithm, named STAMP [1], utilizes a fast subsequence search algorithm based on the Fast Fourier Transformation, called MASS. With it, the distances of one selected subsequence to all others within the length n time series are computed with $\mathcal{O}(n\log(n))$ runtime and linear space complexity. By application of the method to all subsequences, the overall matrix profile can be computed in $\mathcal{O}(n^2\log(n))$ time. An important advantage compared to the brute-force method with $\mathcal{O}(n^2 \cdot m)$ runtime is not only the absolutely iproved runtime, but also the independence from the user-specified subsequence length m. A random ordering of those searches makes it an *anytime* algorithm [31], i.e. the current result state during computation constitutes always a approximate solution, which is quickly converging to the final one. Further an incremental variant for streaming data is presented.

STOMP [2] further improves the runtime complexity to $\mathcal{O}(n^2)$ while preserving the linear memory requirement. For all-pair-similarity joins without pruning of any pairwise computations, this is optimal, as there are n^2 pairs of subsequences to be compared. While also an incremental version for streaming data is shown, realization as an anytime algorithm is not possible.

Zhu *et al.* [32] present with SCRIMP the currently most advanced algorithm: it provides the

same optimal runtime behavior as STOMP and the anytime property. The lower convergence rate compared to STOMP is mitigated by another presented fast algorithm for computation of an approximate matrix profile, called *PreSCRIMP*. By execution of PreSCRIMP in advance to SCRIMP, the fastest anytime convergence rate among all algorithms is achieved. The combination of both algorithms is also referred to as *SCRIMP++*. The PreSCRIMP variant appeals by its very short absolute runtime at the considered scale, while its runtime complexity is only $n^2 \log(n)/s$. The parameter s is a stride for sparse sampling of evaluated distances and proposed to be set to quarter of the search window length $s = m/4$ as a reasonable choice.

2.3 HPC Approaches to Time Series Analysis

In order to speed up time series analysis, various HPC approaches are employed in previous work. This shows the interest in acceleration and scalability towards larger time-series in the field. Many of those works employ different algorithms, which became outdated. Often they are targeted at very specific applications and less versatile than the matrix profile or achieve only limited scalability.

As a first example, the *Parallel Discord Discovery* of Huang *et al.* [33] targets the problem of discord discovery only and utilizes an Apache spark cluster. It is based on the HOT-SAX technique [34], which applies a discretization to the time-series and therefore computes only an approximate solution in contrast to our work. The approach splits the time series into segments, which are distributed among the nodes for comparison against all other time-series parts. The comparison is structured into different rounds, in which bulks i.e. segments of the time-series consisting of several subsequences, are rotating between the nodes. More bulks than workers are created and a work queue is used to improve load balancing. Details of mapping the approach to the Spark system are not reported. Huang *et al.* [33] asses scalability by reporting runtimes, speedup and efficiency on a Spark cluster up to 10 nodes (not stating a core count) and problem sizes in the range of 10^5 to 10^7 sample values. They achieve a fairly constant parallel efficiency of approximately 73 $\%$ in strong scaling. Furthermore they evaluated the quality and parameter setting of their load balancing mechanism, achieving idle time rates down to $\approx 5\%$.

Berard *et al.* [35] target similarity search of user-specified query subsequences with an Hadoop implementation. Reportedly up to 20 cluster nodes were used to speed up computations. Retrieval of either the K-nearest neighbors or all similar sequences within a distance range are implemented. The work could be extended to a full all-pairs similarity join, as targeted in our work with the matrix profile, by subsequent executions in which all the subsequences within a time-series as act as queries. The approach shows a specific weakness, when applying the map-reduce processing scheme to time-series in a database: for storage in the database, the time series is split into consecutive segments, whose length exceeds the user-selected maximum query sequence length. Storage records are created from pairs of neighboring segments by concatenating them to a subsequence, i.e. an array of real values.

Hence each segment is redundantly stored in two database entries: one time as the leading and another time as the trailing half of a subsequence. The storage redundancy is required to perform a sliding window subsequence search. The chosen maximum query length limits applicable queries (without restructuring the database) and impacts the database layout: longer queries require larger data entries and might have impacts on the database performance. The presented solution employs Euclidean distance computations with early abandoning based on the best-so-far found subsequence distance. It was proven to have worse inferior runtime complexity than the approach employed in the matrix profile [1]. For that reason scaling of the approach to large clusters is inefficient. Neither do the absolute runtimes for small problem sizes show gains compared to the approach of [30].

Movchan *et al.* [36] also accelerate the single query nearest-neighbor subsequence search. They rely on the dynamic time warping (DTW) distance with the help of an Intel Xeon Phi CPU and coprocessor, utilizing the OpenMP programming model and outline their interest in extending their work to a cluster system. They compare runtimes to previous similar efforts based on GPUs and FPGAs, demonstrating highly superior performance compared to those. Scalability and quality of their parallelization though is not reported. As they employ DTW, the approach inherits its runtime behavior. The time complexity for all-pairs similarity join of a length n time series is $\mathcal{O}(n^2m)$, which depends on the query window length m in contrast to the (n^2) complexity of the matrix profile approach of Zhu *et al.* [37].

Another parallelization of a fault (discord) detection based on HOT SAX is presented by Loh *et al.* [13] on a multi-core processor. In their specific application domain, they gather several independent time series streams. Thus they choose to concurrently run the subsequence searches on the different time series, limiting the approach to such settings. The reported runtime ratios up to 8 threads exhibit a inefficient parallelization and limited scalability.

A first approach to accelerate the matrix profile computation is GPU acceleration, as presented by Zhu *et al.* [37] along with the efficient STOMP algorithm. The work pushes the limit of maximally feasible time-series length all-subsequence-similarity-joins and demonstrates computation of a matrix profile for a $1 \times 10^8\,\mathsf{sample}$ time series from a seismic data set within 12 days. In contrast to our work, it is limited by the hardware of a single accelerator. Namely the available memory limits the maximally feasible problem size and for larger problems growing runtimes are enforced by the limited speed of the GPU. In contrast to that, cluster systems as considered in our work allow scaling of the computational resources such that for a range of even larger problem sizes reasonable runtimes can be achieved.

During our ongoing work, a preprint of the STAMP framework publication [2] became available. Concurrently to ours and finished a little earlier, they realized a cluster parallelization of the matrix profile computation, justifying our effort. They targeted an Amazon AWS cluster and in contrast to our approach schedule work partitions in a job queue. Intermediate results are merged in a additional final job. In contrast to our work, they also employed GPU acceler-

ators for further speedup and exemplify the scalability of their approach on a 1×10^9 sample time series. We review details of their work for comparison to ours in dedicated sections. Namely we discuss kernel optimizations in 4.4, the parallelization scheme in 5.2.4 and experimental results in 6.4.2.

3. Background

In this chapter we summarize the theoretical background from literature, on which we build in the following chapters. In particular the necessary mathematical definitions for the matrix profile are stated in section 3.1. The SCRIMP algorithm for its computation, which our work builds on are reviewed in section 3.2.1. Readers familiar with the algorithm might skip the chapter and only use it as a reference, as we adopt the nomenclature of Zhu *et al.* [37]. The terms, definitions and techniques we use for investigation of the performance of our implementation are composed in section 3.3. Readers familiar with high performance computing might skip the section and only look up terms if needed.

3.1 Definitions

3.1.1 The Matrix Profile

Let us first formally define the necessary terms, followed by a brief informal explanation. In this work we will stick to the nomenclature of Zhu *et al.* [37]. In particular we utilize following definitions from their work:

Definition 1 *A **time series** T of length $n \in \mathbb{N}$ is a sequence of real numbers $t_i \in \mathbb{R}$:* $T = t_1, t_2, \ldots, t_n$

Definition 2 *Given a time series T of length n, a **subsequence** $T_{i,m}$, is a contiguous subsequence of T of exactly $m \in \mathbb{N}$ elements starting with element $i \in [1, 2, \ldots, n-m+1]$:* $T_{i,m} = t_i, t_{i+1}, \ldots, t_{i+m-1}$

The definition of the time-series (def. 1) is straight forward. The most notable fact is that (in theory) we are dealing with values of a continuous range, which are sampled at discrete time points with a typically fixed sampling rate. The most important part of the definition is, how

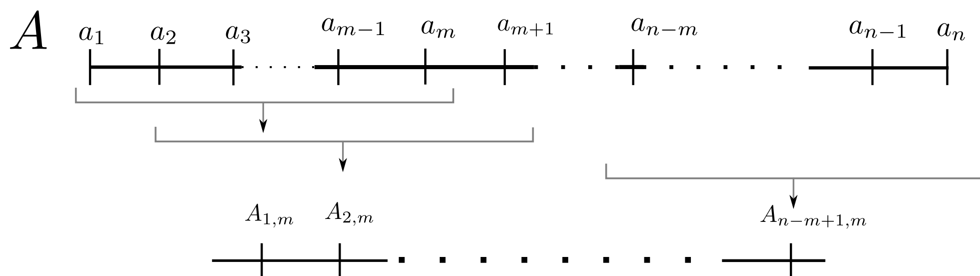


Figure 3 Time series A of length n and its subsequences of length m

the starting point and length of subsequences are denoted. It is remarkable, that storing a time series as a contiguous vector of values is inherently a memory efficient way of storing all its subsequences. Figure 3 illustrates, how a pair of subsequences with consecutive starting indices overlap. This also makes clear, how A of length n decomposes into exactly $n - m + 1$ subsequences of length m. By definition obviously subsequences are time-series themselves.

In order to deal with all cases, including similarity joins between two distinct time series as in [1], we further employ slightly modified definitions, which finally yield the same semantics of the matrix profile.

Definition 3 *Let A and B be two time series of lengths $n \in \mathbb{N}$ and $l \in \mathbb{N}$, which are either identical or do not overlap. Given a query length $m \in \{x | x \in \mathbb{N}, 2 < x \leq \min(n, l)\}$ the* ***distance matrix*** *$D_{AB} = (d_{i,j})_{l-m+1,n-m+1}$ defined by $d_{i,j} = \delta(B_{i,m}, A_{j,m})$, where $\delta : \mathbb{R} \times \mathbb{R} \to \mathbb{R}_0^+$ denotes the z-normalized euclidean distance (which will be explained in section 3.1.2)*

Utilizing the "z-normalized euclidean distance" metric, the distance matrix D_{AB} contains all pairwise distances between subsequences from A and B (Illustrated in figure 4). The special case, in which we substitute them for a single series T is remarkable, as performing a self-join of a single time series is the typically considered case in most of the matrix profile publications [21], [22], [27], [37]. In this case, the distance matrix D_{TT}, which we will abbreviate with D, has the important property to be symmetric. This can be exploited to save most of the computations. In addition to the symmetry, diagonal entries $d_{i,i} = 0$ are observed, as they measure the distance between a subsequence and itself (because the z-normalize euclidean distance is a proper metric). Furthermore, close to the diagonal typically all distance values will be quite small, as close by subsequences $S_{i+x,m}$ (for small $x \in \mathbb{N}$) of a sequence $S_{i,m}$ typically are still quite similar, i.e. have small distance values. Considering the objective of searching similar subsequences, such matches are called *trivial*. A formal definition can be found in [38] or [18], a visual explanation for example in [27]. Trivial matches can be observed in any case, where the two analyzed time series A and B overlap by more than m time points (in the wall clock time of the value recording).

Definition 4 *In addition to all the prerequisites and contents of definition 3, let $E_i \subset [1; k - m + 1]$ denote the set containing the indices of all trivial matches of the subsequence $B_{i,m}$ in A. Let the* ***matrix profile*** *P_{AB} be defined as the vector $P_{AB} = (p_1, p_2, ..., p_{l-m+1})$ where $p_j = \min_{i \in [1;k-m+1] \setminus E_i} d_{i,j}$*

Definition 5 *Given the prerequisites and contents of definition 4, let the* ***profile index*** *be defined as the vector of indices $I_{AB} = (q_1, q_2, ..., q_{l-m+1})$ with $q_j = \arg\min_{i \in [1;k-m+1] \setminus E_i} d_{i,j}$. In the (theoretical) case of several minimizers j, the smallest one shall be chosen.*

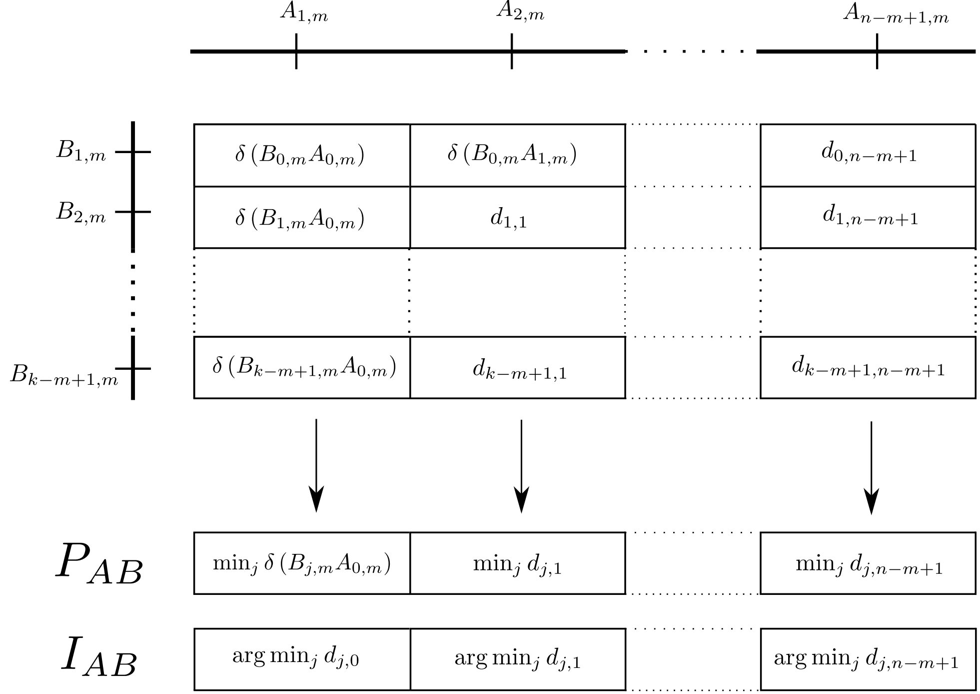


Figure 4 General definition of the matrix profile P_{AB} and profile index I_{AB}. The pairwise distances $d_{i,j} = \delta(B_{i,m}, A_{j,m})$ between all subsequences of the time series A and B form the distance Matrix D_{AB}. The minimum distance and the minimizing index of a column form the entries of the matrix profile P_{AB} and the profile index I_{AB}.

Definitions 4 and 5 formally state our objective data-structures, the matrix profile and profile index. Disregarding the exclusion zone E_i of trivial matches, the matrix profile denotes a vector whose elements are the minima of the columns in the distance matrix. As figure 4 illustrates, the i-th entry of the matrix profile P_{AB} denotes the minimum among the distances between subsequence $A_{i,m}$ and all $B_{j,m}$ with $1 < j < k - m$, where k is again the length of B and m a chosen subsequence length. Informally the entry $P_{AB,i}$ of the matrix profile is the distance of $A_{i,m}$ to its nearest neighbor among all the subsequences of B with length m. The profile index I_{AB} captures the starting index j of this nearest neighbor.

The effect of the exclusion zone E_i is to avoid trivial matches. As explained previously, trivial matches occur, in case that A and B overlap. In order to simplify treatment of trivial matches (and capturing the two most important use-cases), we restrict ourselves in definition 3 to the two cases of either non-overlapping or identical series A and B. In the non-overlapping case there exist no trivial matches, hence we can set $E_i = \emptyset$. In case a self-similarity search on a single time series T is performed, as depicted in figure 5, values along and close to the diagonal of the distance matrix need to be excluded from the search. Hence only the shaded lower and upper triangles are considered for computing the matrix profile. In [37] this is achieved by ignoring a fixed number of elements around the self-match, i.e. setting $E_i = [i - m/4; i + m/4]$. The approach ignores the fact, that in theory the size of the trivial matching zone is data-dependent, as illustrated by Yeh *et al.* [27]. Though it can be reasoned, that for applications like motif discovery (see sec. 2.2), one does not expect overlap of more

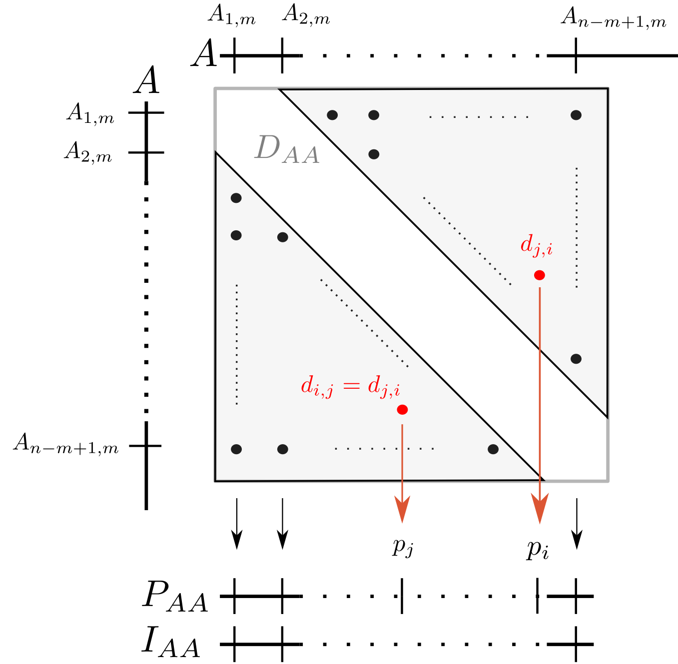


Figure 5 Matrix Profile in case of a self-similarity search. Dots indicate relevant entries of the the distance matrix D_{AA}. Only distances in the shaded triangular region of the matrix are considered for computation of the matrix profile P_{AA} and its index I_{AA}. The white part along the diagonal depicts the exclusion zone. Symmetry of the matrix can be exploited, by calculating distance values $d_{i,j}$ only in one of the shaded triangles and updating the profile and index at the two locations i and j at the same time.

than 75% of relevant subsequences [27].

Figure 5 also reveals, how the symmetry can be exploited in the case of a self-similarity search: It suffices to compute distances $d_{i,j}$ in the lower relevant triangle. The value is then checked for determining the matrix profile and index at locations i and j. The definitions also reveal the nature of the matrix profiles' naming: The matrix profile captures essential information of the distance matrix.

3.1.2 Z-Normalized Euclidean Distance

In order to measure similarity between time series subsequences, all matrix profile algorithms rely on z-normalized euclidean. We review the definition and justification of its usage in the next two subsections.

Definition

Let $A = a_1, \dots a_m$ and $B = b_1, ...b_m$ both be time series of the same length $m \in \mathbb{N}$. With $\hat{A}$ we denote the z-normalized version of A, i.e. a rescaled version with zero mean and unit variance. Denoting the sample mean and standard deviation of A with μ_A and σ_A, it is

formally defined as:

$$\hat{A} \coloneqq \frac{a_1 - \mu_A}{\sigma_A}, \ldots, \frac{a_m - \mu_A}{\sigma_A} \tag{3.1}$$

The z-normalized euclidean distance $\delta(A, B)$ of A and B is just the euclidean distance between their z-normalized versions [21]:

$$\delta_{A,B} \coloneqq \sqrt{\sum_{i=1}^{m} (\hat{a}_i - \hat{b}_i)^2} \tag{3.2}$$

While this formulation gives a nice interpretation, for computation in the matrix profile algorithms it is useful to employ a different formulation [21]:

$$\delta_{A,B} = \sqrt{2m \left(1 - \frac{Q_{A,B} - m\mu_A\mu_B}{m\sigma_A\sigma_B}\right)} = \sqrt{2m\,(1 - corr(A, B))} \tag{3.3}$$

Here for brevity $Q_{A,B}$ denotes the dot product (interpreting the time series as vectors) of A and B : $Q_{A,B} = \sum_{i=1}^{m} a_i \cdot b_i$. The formula relates the distance to the correlation coefficient $corr(A, B)$. While this might be mathematically interesting, we will not make further use of that fact. A derivation of the equation can be found at [39].

Justification

The definition of the matrix profile, relying on the defined distance matrix (Definition 3), explicitly relies on the *z-normalized Euclidean distance* as the metric between subsequences. It would be possible to generalize the definition to arbitrary similarity metrics. Its usage is empirically justified by Wang *et al.* [40] in a comparison of known similarity metrics for 1-nearest-neighbor classification . It was found, that for larger data sets the error rates converge and that more complex distance measures do outperform Euclidean distance only on specific data sets. The gains of z-normalizing the data (or considering the z-normalized Euclidean distance) were highlighted in [41] and illustrated by [42]. We want to critically state that, while generally being justified e.g. by sensor drift depending on external factors like temperature, the gains are also application dependent and normalization might also yield errors in the case, that subsequences of the same shape but with different amplitudes are considered as different events in a specific application.

3.2 Algorithms

Three algorithms to compute the matrix profile had been published in literature. They have in common, that all rely on z-normalized euclidean distances according to equation 3.3 and enumerate all entries $d_{i,j}$ of the distance matrix (def. 3) without storing the complete matrix in

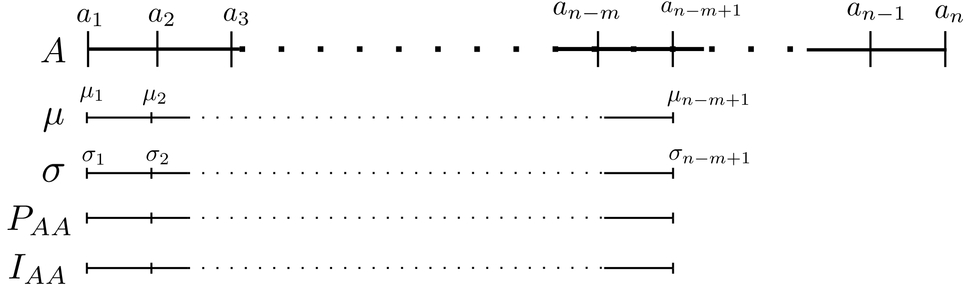


Figure 6 Data layout of the most important data during the matrix profile computation: The time series A itself with length n and its accompanying meta time-series of length $n-m+1$, where m denotes the query length: The sliding mean and standard deviation μ and σ are precomputed before the evaluation of the distance matrix in order to avoid redundant computations. The matrix profile P_{AA} and Profile index P_{AA} are the result buffers updated during the evaluation

memory. They differ in the order of evaluation of the distance matrix entries and in the computation of the dot products $Q_{i,j}$. STAMP [1] exploits the fast Fourier transform to compute the dot products for whole rows or columns of the distance matrix and therefore operates row by row, ordered randomly. STOMP [37] and SCRIMP [32] both compute the distances based on a dependency $Q_{i,j} = f(Q_{i-1,j-1})$. For that purpose STOMP buffers intermediate results and proceeds row by row. In contrast SCRIMP adapts the iteration scheme to the dependency and iterates along diagonal direction in the matrix, with a random order of the diagonals. SCRIMP++ [32] combines the SCRIMP algorithm with an approximate algorithm, called PreSCRIMP, which is run in advance. Its purpose is to get better approximate solutions in the case of early stopping of the algorithm, which is valid due to its anytime property.

We base our work on the SCRIMP algorithm. Like STOMP it has the optimal runtime complexity of $\mathcal{O}(N^2)$ for exact algorithms but does not require buffering a large number of intermediate dot product results. It performs a exact all-pairs-similarity join and relies on a single user specified parameter. We provide the details of the algorithm in the next section.

3.2.1 SCRIMP

Precomputations and data structures

SCRIMP [32] compares subsequences based on z-normalized euclidean distances, computed according to equation 3.3. It requires the means μ_x and σ_x of subsequences starting at time point t. As all subsequence pairs are considered, a individual μ_i and σ_i will be required several times, specifically once per entry of the matrix profile. To save redundant computations, Zhu [32] proposed to compute and store those values in advance to the evaluation of the distance matrix. As those values exist for each subsequence $A_{i,m}$ of a fixed length m, they can be considered as meta-time-series of length $n-m+1$ where n denotes the length of times series A. As explained in 3.1.1, the matrix profile and profile index are series of the same length and can also be considered as meta time series. Figure 6 summarizes all those data series. The length is depicted logically, i.e. depicting the amount of entries for each value series. Except for the profile index I_{AA}, which stores indices and therefore integers,

the remaining meta-time series are all real values and stored as floating point values. Overall there are $(n-m+1)\cdot 4+m-1$ floating point and $n-m+1$ integer values required. Neglecting small intermediate data-structures, the upper bound for the memory consumption is therefore $\mathcal{O}(n)$. Note, that the actual distance matrix is not stored in memory in closed form. It is only evaluated in an iterative scheme, which utilizes the presented data. While the illustration depicts the data as several arrays, of course it is also open for a implementation to choose a different layout, for example a array of structures or some mixture.

Iteration scheme

SCRIMP [32] matches the iteration scheme to the relation between dot products $Q_{i,j}$ and $Q_{i-1,j-1}$ which are neighboring in diagonal direction. Given the problem of a self-join on a time series $A = a_1, .., a_n$ it holds for dot products $Q_{i,j}$ of the subsequences $A_{i,m}$ and $A_{j,m}$, that:

$$Q_{i,j} = Q_{i-1,j-1} + a_{i+m-1}\cdot a_{j+m-1} - a_{i-1}\cdot a_{j-1} \tag{3.4}$$

In contrast to a implementation of the dot product definition $Q_{i,j} = \sum_{x=0}^{m-1} a_{i+x}\cdot a_{j+x}$, which would require $2m-1$ floating point operations, this formulation has the constant cost of four FLOPs. The low amount of computations comes at the cost of a dependency $Q_{i,j} = Q_{i-1,j-1}$ in diagonal direction. SCRIMP adopts the dependency directly into its iteration scheme, as illustrated in figure 7: Initially all the dot products $Q_{i,1}$ corresponding to distances in $d_{i,1}$ in the first column of the distance matrix are computed. This can be done either by a naive dot product implementation or more efficiently with the help of MASS. A entry $d_{k,1}$ which had not been considered so far is chosen at random as a starting point. The distance $d_{k,1}$ is computed by formula 3.3 and the matrix profile is updated, if a new minimum value is found. Iterating over consecutive $x = 1, 2, ...$ the dot products $Q_{k+x,1+x}$ and distances are computed according to formulas 3.4 and 3.2 respectively. Visually this iteration proceeds along diagonal direction, as depicted by the straight red arrows in figure 7. After the last entry $d_{n-m+1,n-m+1-k}$ of a diagonal was processed, a new starting index k, which had not yet been processed, is chosen and the next diagonal is evaluated the same way.

3.3 Performance of Parallel Programs

Depending on the application scenario, various performance requirements for parallel programs are of interest. Hwang *et al.* [43, p. 102] names for example *speed*, *throughput*, *utilization*, *cost-effectiveness* and *performance/cost ratio*. The performance of a parallel program depends not only on the properties of the parallel algorithm but also on the hardware architecture and software environment [44, pp. 169 sqq.], [43, p. 12]. For this reason the whole *parallel system* needs too be considered. While theoretical analysis of algorithms can model some system specifics like network architecture [44, pp. 184 sqq.][45], in practice empirical measurements are conducted to asses the performance.

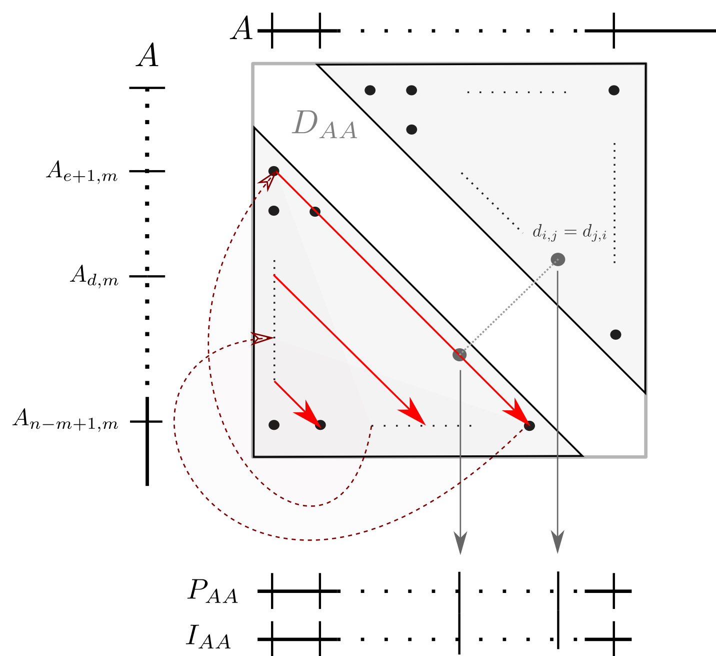


Figure 7 Scrimp iteration scheme for a self-similarity search on a time series A: The entries of the distance matrix are computed and evaluated in a two dimensional iteration scheme: The inner iteration proceeds along diagonals, as depicted by the red arrows: starting from a entry $d_{k,0}$ successively $d_{k+1,1}, d_{k+2,2}, \ldots d_{n-m+1,n-m+1-k}$ are considered. The outer loop iterates over all different diagonals in the lower triangle of the distance matrix, except for the exclusion zone. Exploiting the symmetry of the distance matrix, two matrix profile updates are considered at each entry evaluation. This way the iteration over the upper triangle is omitted.

Of particular interest for high performance computing systems with large numbers of processing elements is so called *scalability*: Rauber *et al.* [44] state it generally as the ability to increase specific performance values like the amount of computations or productivity proportional to added parallel compute resources. Literature contains various definitions for the term and the publication provides a good starting point for interested readers beyond the scope of this thesis. To analyze scalability in our work we adopt the metrics listed in Quinn [46] which we review in subsection 3.3.2.

Highest computational performance on parallel machines is only achieved, if the program makes efficient use of the available processing hardware features. Further high scalability on parallel machines is easier to achieve for computations running slowly and therefore such investigations are only valid, if the performance of the computational kernels is close the the maximum hardware limit. Subsection 3.3.1 presents the theoretical background of methods we use for investigation of the sequential kernel performance.

3.3.1 Kernel Performance

Roofline diagrams

Optimizing the performance of a kernel for a particular computing hardware requires knowledge about the hardware limitations as well as the resource utilization of a program to reason about potential improvements. Detailed information about hardware limits like maximum memory bandwidths could be obtained for example with benchmarks like STREAM [47]. Hardware utilization of program sections can be analyzed by application profiling together with hardware simulation [48] or tracked at the highest level of detail with performance counters and source code instrumentation APIs like PAPI [49].

Roofline charts are a visual way to summarize information about the application behavior and machine limitations. In contrast to the previously outlined manual effort, roofline analysis with tools like the Intel Advisor [50] collect information from micro-benchmarks, application profiling and hardware counters in a unified framework. For analysis of our work we rely on the *Cache-Aware-Roofline model* (CARM) as introduced by Ilic *et al.* [51], in contrast to the original roofline model [52] and the more recently proposed *integrated roofline* [53], which provides more information but was not yet available to us.

Figure 8 shows a example of a CARM chart. Target of investigation is the performance of loops in the application. Relevant loops are selected based on the time spent in them and show up as individual data-points like *C* and *M* in the illustration. The achieved compute performance in GFLOPS within loops is plotted on the vertical axis. The horizontal one shows the *arithmetic intensity*, abbreviated as *AI* and both axis are plotted in logarithmic scale. The arithmetic intensity is the ratio of floating point operations performed in the loop to the amount of data requested from the CPU core. The AI refers to data transfers between the CPU and any part of the memory sub-system independent of the potential cache level at which the data resides. For computational kernels in a program, the AI is mostly fixed by the equations

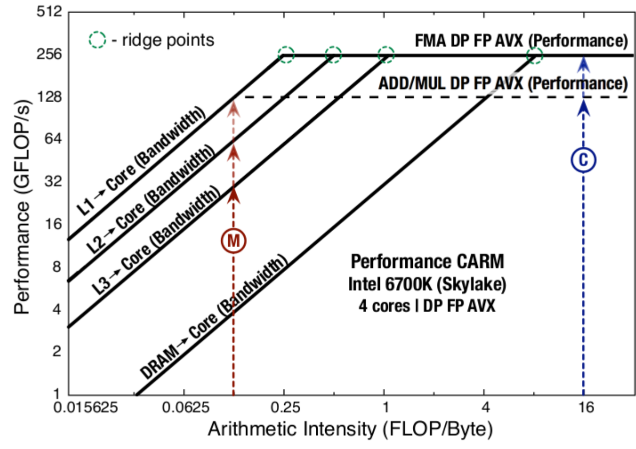


Figure 8 Example of a cache-aware-roofline diagram with explanatory annotations by Marques *et al.* [50]

of the algorithm and therefore constant. For this reason kernel data points can move in the CARM only in vertical direction if the arithmetic within a kernel is unmodified.

Highest achievable compute performance of loops has a upper limit given by the peak performance of the machine or by the lowest level (L1) cache bandwidth, as illustrated by the points *C* and *M* in figure 8 respectively. Additionally the achieved performance might be limited by bandwidths of lower levels of the memory hierarchy, if data loaded into the core from such. Such hardware limitations obtained from micro-benchmarks are illustrated as so called *rooflines* in the CARM.

Comparing the location of kernel loops to the rooflines can indicate whether a kernel is limited by a certain memory hierarchy bandwidth, a inefficient or lack of instruction-level-parallelism or the machines peak compute rate. Accordingly potential optimizations can be considered.

The kernel location within the roofline diagram can only indicate, but not proof, limiting bottlenecks, except for the case that the kernel ends up at the topmost roofline. Sub-optimal computed rates can also be caused for example by memory and instruction latencies, branch misprediction or address-virtualization [54] and potentially further investigation of bottlenecks is necessary.

Instrumentation

Modern CPUs contain *performance monitoring units*, which provide configurable counters for hardware events like the number of executed operations, cache or branch mispredictions. Such counter values can be used to detect hardware bottlenecks which prevent applications to achieve the machines peak-performance. Profiling tools like *GNU perf* [55] can be used to obtain such counter values at application or call-path level. Manual source code instrumentation with programming interfaces like PAPI [49] or the LIKWID marker API [56] allows

developers to instrument their application code with specific measurement section to record counter values. While the effort for source code instrumentation is highest compared to other methods, it allows to obtain event counts and performance metrics for arbitrary program sections and can potentially achieve the lowest runtime overhead, if only few measurements are required. Low runtime overhead of such measurements is important, as increased runtimes due to profiling or instrumentation can mask performance bugs and render the measurements pointless.

In our work we adopt source code instrumentation with the LIKWID marker API [56] to examine the sequential performance of computational kernels. In contrast to others the API is particularly appealing by the fact that the desired counters are selected at program invocation time, once the instrumented application is compiled. Switching the counters of interest does not require recompilation. Further, predefined sets of counters for supported hardware, called *event groups*, are provided to measure typical subsystem behaviors like cache misses.

3.3.2 Scalability

Metrics

High performance parallel processing of computational tasks is motivated by two correlated aspects of the computation: the amount of work and the required program runtime for processing it. The used amount of parallel processing elements p links the two dimensions to each other.

The amount of work, or *problem size*, k can be measured for computational tasks in terms of the arithmetic operations required in a sequential program and can also be stated relative to a base problem size [57]. Central metric for performance analysis is the *parallel execution time* $T_{\mathrm{par}}(p, k)$. It is defined as the time from start of the first to the end of the last process in the application and can be measured in applications or with external tools. The special case of time T_{sequ} taken for execution on a single processor is called *sequential execution* or *sequential run time*. Further, the total processing time or *cost* of the parallel program execution with p processing elements is given by:

$$T_{\mathrm{total}}(p, k) = p \cdot T_{\mathrm{par}}(p, k) \tag{3.5}$$

In contrast to a sequential software, a parallel program contains additional execution times due to the parallelization, for example time for data exchange or waiting times due to synchronization. Such runtimes are called *parallel overheads*. The *total parallel overhead* of a program execution with p processes and problem size k is obtained by:

$$T_{\mathrm{o,total}}(p, k) = p \cdot T_{\mathrm{par}} - T_{\mathrm{sequ}} \tag{3.6}$$

Runtime overheads are of particular interest for parallel performance analysis, as they can impose constraints like minimum parallel execution times and limit scalability.

Typically two *scaling* scenarios for parallel programs are considered. The first scenario is increasing the amount of used processing elements p to reduce the parallel runtime for a fixed problem size k_0, which is called *strong* or *fixed-size* scaling. It is particularly relevant for large computational problems, whose sequential processing time is too large for practical application. In the second scenario, *weak scaling*, the computational workload is increased at the same rate as the number of processing elements, such that always $k \propto p$ holds. It is of particular interest for applications in which results benefit from increased detail.

A series of similar metrics in both scenarios can be used to compare program behaviors in such scaling scenarios. Comparison is possible either with each other or to idealized models. The metrics can be computed from empirical runtime measurements as follows [57]:

strong scaling

setting $k = k_0 = \text{const}$

speedup:

$$S(p) = \frac{T_{\text{par}}(1, k_0)}{T_{\text{par}}(p, k_0)} = \frac{T_{\text{sequ}}}{T_{\text{par}}(p, k_0)}$$

parallel efficiency:

$$E(p) = \frac{T_{\text{par}}(1, k_0)}{p \cdot T_{\text{par}}(p, k_0)} = \frac{S(p)}{p}$$

(empirical) serial fraction

$$f(p) = \frac{1/S(p) - 1/p}{1 - 1/p}$$

weak scaling

setting $k = p \cdot k_0$

scaled speedup:

$$S_k(p) = \frac{p \cdot T_{\text{par}}(1, k_0)}{T_{\text{par}}(p, p \cdot k_0)} = \frac{p \cdot T_{\text{sequ}}}{T_{\text{par}}(p, p \cdot k_0)}$$

parallel efficiency:

$$E_k(p) = \frac{T_{\text{par}}(1, k_0)}{T_{\text{par}}(p, p \cdot k_0)} = \frac{S_k(p, p \cdot k_0)}{p}$$

scaled serial fraction

$$f_k(p) = \frac{1/S_k(p) - 1/p}{1 - 1/p}$$

Idealized behavior of fully scalable programs without any parallel overheads show speedups of $S_{(k)}(p) = p$, efficiencies of $E_{(k)}(p) = 1.0$ and have zero serial fractions $f_k = 0$. Overheads in programs cause reduced speedups and efficiencies but increase serial fractions.

In both cases, the serial fractions measure a fraction of the total sequential processing time T_{sequ} which is not parallelized. In both scaling scenarios, parallel systems with constant $f_{(k)}$ form a set of theoretical models, whose speedup saturates at a upper limit towards large numbers of processors. Those upper limits can derived in theory according to laws called Amdahl's and Gustafson's law [46]. In practice, overheads in parallel programs rarely follow such ideal behavior and the respective models are not applicable. Often communication overheads grow with the numbers of processors such that speedups are decreasing beyond a peak point [58].

Investigating scalability with the help of the described metrics based on experimental measurements alone is of limited use only: because measurements sum up different code parts, the observed behavior is dominated by code parts, in which most runtime is spent at the scale of the chosen problem sizes and processor numbers. It is possible, that other code

parts with high runtime complexities require very small amounts of time at the investigated scale but will be dominating with further increase of the parameters. Theoretical analysis like the isoefficiency metric are suited to predict such behavior and compared algorithm variants by theoretical modeling of overheads. With a implementation at hand, profiling of the application can provide empirical insight at function-level.

Profiling

Calotoiu *et al.* [59] provide with Extra-P a tool to investigate the scaling behavior of individual functions in applications. Profiling data collected in scaling experiments with the Score-P measurement infrastructure [60] is used to select the best fitting model from a parametric set of functions. This allows developers to detect unscalable code parts by investigating the long-term-dominating functions based on the fitted scaling models.

The hypothesis for measured runtimes of (exclusive) time spent in functions are of *performance model form*, which is given for a single parameter p by:

$$f(p) = \sum_{k=1}^{n} c_k \cdot p^{i_k} \log_2^{j_k}(p) \tag{3.7}$$

The coefficients $i_k, j_k \in \mathbb{Q}$ are initially chosen from a predefined set and refined in the iterative fitting-process. In case of multi-parameter fitting, for example fitting to the problem size and processor number, additional factors of the same form are added. Selection of models is performed in a iterative process with increasing n using cross-validation to avoid over-fitting. Model fitting is judged based on *adjusted* R^2 coefficients of determination, symmetric mean average percentage error (*SMAPE*), and the residual sum of square (*RSS*) error.

We chose to apply the same metrics for fitting of theoretical models to our experiment data. Let $f : \mathbb{R} \to \mathbb{R} : x \mapsto f(x)$ denoted the fitting hypothesis with a number of k parameters, n the number of measurements and $\forall i \in 1, \ldots, n (x_i, y_i) \in \mathbb{R}^2$ a set of data points. The fitting metrics are defined as

$$\mathrm{RSS} = \sum_{i=1}^{n} (y_i - f(x_i))^2 \tag{3.8}$$

$$R^2 = 1 - \frac{\mathrm{RSS}}{\sum_{i=1}^{n} (y_i - \bar{y})} \tag{3.9}$$

$$\text{adjusted } R^2 = 1 - [\frac{(1 - R^2)(n-1)}{n-k-1}] \tag{3.10}$$

$$\mathrm{SMAPE} = 100\,\% \frac{\sum_{i=1}^{n} |f(x_i) - y_i|}{\sum_{i=1}^{n}((f(x_i) + y_i))} \tag{3.11}$$

Isoefficiency analysis

Another way to investigate scaling behavior of parallel algorithms are theoretical studies of the algorithm. Runtime behavior of individual program sections, like global communication, can be modeled analytically and used to investigate the presented scaling metrics from a theoretical point of view. Carmona *et al.* [61] for example propose such a analysis with a unified view of both scaling scenarios. For our work we rely on the isoefficiency metric [62]. It allows to compare algorithms based on a single scaling complexity function and is well adopted in literature. [44].

The isoefficiency metric is given by the rate at which the problem size needs to be increased to maintain constant parallel efficiency while the number of processors is increased. Only parallel systems for which such a rate exists are considered as *scalable*. Lower growth rates indicate higher scalability of parallel systems. Systems with the lowest possible isoefficiency function of $\Theta(p)$ are called ideally scalable.

Computing the isoefficiency function requires to compose an analytical model for the complexity of the total parallel overhead $T_{\mathrm{o,total}}(p, W)$, where W denotes the problem size (we adopt the notation of [61] for the discussion in this section). The total overhead is decomposed into a sum of individual terms $o_i(p, W)$: $T_{\mathrm{o,total}}(p, W) = \sum_i o_i(p, W)$. All those contributions are balanced against the problem size with proportionality constants K_i:

$$W_i(p) = K_i \cdot o_i(p, W_i(p)) \tag{3.12}$$

From all resulting functions $W_i(p)$, the isoefficiency metric is obtained as the dominating growth rate. A full analysis needs to consider one more growth term W_c based on the so called *degree of parallelism*, i.e. the maximum possible numbers of processes usable due the algorithm structure. We omit the explanation, as it does not impact our work.

4. Sequential Optimization

Algorithms need to be optimized to achieve highest hardware utilization of the single cores of high performance systems. Otherwise computation is uneconomical and parallel scaling results are biased. In this chapter we present a series of sequential optimizations for the matrix profile computation: In section 4.1 arithmetic rearrangements of the kernel are presented, which reduce the amount of required computations. In section 4.2.2 a modified iteration scheme is presented to avoid a memory bottleneck on the machine. For full usage of the hardware capabilities, in section 4.3 we briefly discuss vectorization of the approach. Finally we compare our sequential optimizations to those presented in the preprint of the SCAMP framework [2], which was published concurrent to our work.

Starting point of our forms the published SCRIMP C++ kernel [63] of Zhu *et al.* [32].

4.1 Arithmetic Kernel

As presented in section 3.1.2, the matrix profile algorithms compare subsequences based on the euclidean distance kernel. It is given as [37]:

$$d_{i,j} = \sqrt{2m\left(1 - \frac{Q_{i,j} - m\mu_{A,i}\mu_{B,j}}{m\sigma_{A,i}\sigma_{B,j}}\right)} \tag{4.1}$$

In this formulation m is the constant query window length, $\mu_{T,x}$ the mean over the values of subsequence $T_{x,m}$ of length m starting at position x and $\sigma_{T,x}$ the standard deviation of the values in the same window. $Q_{i,j}$ is the dot product of the two time series subsequences $B_{j,m}$ and $A_{i,m}$ of length m starting at i and j respectively. It can be defined using the elements $a_1, a_2, \ldots$ and $b_1, b_2, \ldots$ of the input series as:

$$Q_{i,j} = \sum_{k=0}^{m-1} a_{i+k} \cdot b_{j+k} \tag{4.2}$$

The computation according to this formula is inefficient. Different computation schemes for the dot products $Q_{i,j}$ constitute the difference between the STAMP and STOMP/SCRIMP algorithms, as explained later on. As stated by Zhu *et al.* [37], the kernel 4.1 needs to be evaluated $\Theta(N^2)$ time, where N denotes the time series length. In contrast to that, only $\Theta(N)$ different μ_i an σ_i exist. For this reason it is more efficient, to compute and buffer those in advance to the iteration over the distance matrix.

When considering the overall objective, namely computation of the matrix profile and the profile index, this kernel can be further simplified. Denoting the i-th entry of the matrix profile

with p_i and the corresponding profile index as I_i, the objective is to determine for all i:

$$p_i = \min_j d_{i,j} \tag{4.3}$$

$$I_i = \arg\min_j d_{i,j} \tag{4.4}$$

For brevity only the profile value p_i is considered in further discussion. The index I_i stores the position of the found distance minimum and can not be optimized. As the square root is strictly increasing and $2m$ is constant, the profile can be reformulated:

$$\begin{aligned} p_i = \min_j \sqrt{2m - 2\frac{Q_{i,j} - m\mu_{A,i}\mu_{B,j}}{\sigma_{A,i}\sigma_{B,j}})} \\ = \sqrt{2m + \min_j \left(-2 \cdot \frac{Q_{i,j} - m\mu_{A,i}\mu_{B,j}}{\sigma_{A,i}\sigma_{B,j}}\right)} \\ = \sqrt{2m - 2 \cdot \max_j \frac{Q_{i,j} - m\mu_{A,i}\mu_{B,j}}{\sigma_{A,i}\sigma_{B,j}}} \\ =: \sqrt{2m - 2 \cdot \max_j k_{i,j}} \end{aligned} \tag{4.5}$$

Here we introduced a reduce kernel $k_{i,j}$, which requires less computation. Instead of directly evaluating the distance profile, it suffices to consider a simplified profile:

$$\widetilde{p_i} = \max_j k_{i,j} \tag{4.6}$$

$$I_i = \arg\max_j k_{i,j} \tag{4.7}$$

Compared to the original matrix profile, this saves the the costly evaluation of a square root. Still, the original matrix profile is easily obtained in a single pass over the result array in $\Theta(n)$ runtime. It can also be done when storing the final result as follows:

$$p_i = \sqrt{2m - 2\widetilde{p_i}} \tag{4.8}$$

The modified kernel $k_{i,j}$ can be computed most efficiently by appropriate grouping of computations:

$$\begin{aligned} k_{i,j} = \frac{Q_{i,j} - m\mu_{A,i}\mu_{B,j}}{\sigma_{A,i}\sigma_{B,j}} \\ = Q_{i,j} \cdot \frac{1}{\sigma_{A,i}} \cdot \frac{1}{\sigma_{B,j}} - \frac{\mu_{A,i}\sqrt{m}}{\sigma_{A,i}} \cdot \frac{\mu_{B,j}\sqrt{m}}{\sigma_{B,j}} \\ =: Q_{i,j} \cdot s_{A,i} \cdot s_{B,j} - \widetilde{\mu}_{A,i} \cdot \widetilde{\mu}_{B,j} \end{aligned} \tag{4.9}$$

In this formulation, we introduced the new factors $s_{T,n}$ and $\widetilde{\mu}_{T,n}$. As those are combinations of the sliding mean $\mu_{T,n}$, standard deviation $\sigma_{T,n}$ and the constant m only, they can also be

precomputed in a single pass over the time series in $\Theta(N)$ runtime complexity. The original mean and standard deviations $\mu_{T,i}$ and $\sigma_{T,i}$ are replaced by those new meta-time-series which are computed and stored before the matrix evaluations. This rearrangement of the kernel removes many arithmetic operations. Also it avoids utilization of divisions or square roots in the profile kernel, which are more expensive than multiplications.

4.2 Algorithmic Kernel

4.2.1 STOMP/SCRIMP

To complete the definition of the matrix profile kernel, for the computation of equation 4.9, the value of the correlation $Q_{i,j}$ needs to be provided and the matrix profile needs to be updated. We state the kernel in algorithm 1 in a generalized form of a $ABBA$-join, as depicted in figure 9a, because it will be required for the parallelization scheme in section 5.2. The generalized $ABBA$ kernel computes both, the AB-join and the BA-join of two input time series A and B. We will refer to P_{BA} and I_{BA} later on as the vertical result, according to their vertical orientation in the illustration and call P_{AB} and I_{AB} the horizontal one accordingly.

The kernel definition for the self-join on a single time series A is just a special case of this general definition, as illustrated in figure 9b. The single time series A is used for both inputs of the general kernel. Due to the symmetry of the self-join a single result buffer is used for both, the horizontal and vertical result. Doing so, as mentioned earlier, the iteration-scheme can omit kernel invocations for the upper triangle in the distance matrix. Furthermore the iteration scheme needs to omit evaluation of matrix entries in the exclusion zone along the main diagonal.

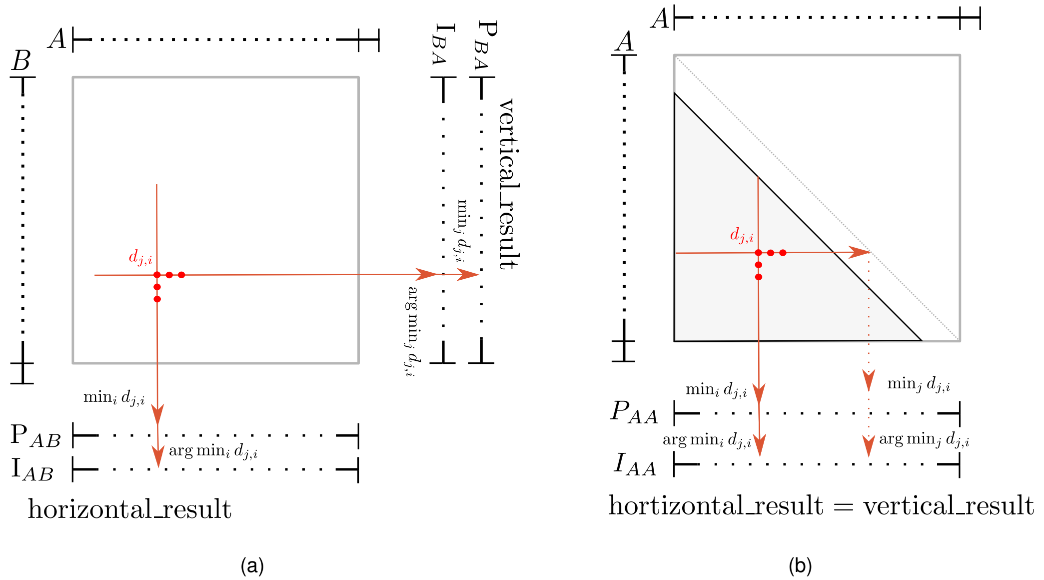


Figure 9 Inputs and outputs of the algorithmic kernel: (a) depicts a generalized $ABBA$ kernel, computing the matrix profiles of an AB-join and BA-join of two distinct time series A and B. The kernel can be reused for computation of a AA self-join as shown in (b), by using a single output buffer for the horizontal and vertical result while providing a single time series A to both input.

As stated in algorithm 1, in the kernel of the STOMP and SCRIMP algorithms [1], [37] the dot product value $Q_{i,j}$ is computed based on the dot product of the upper left neighbor value along the diagonal $Q_{i-1,j-1}$. If such a neighbor does not exist, it can still be obtained by its definition 4.2. Furthermore, in case that a new extremal score value was found, the matrix profiles might need to be updated, i.e. the profile value and the profile index. For this purpose the entries of the vertical and horizontal result buffers are checked at locations according to the row or respectively column coordinate of the evaluated distance value. If it is required they are updated, as formally stated in algorithm 1. For simplicity, the listing assumes equal-length input series, as this is sufficient for our work but it can be generalized for arbitrary lengths.

Algorithm 1 Computational kernel for a similarity-join of time series A and B as in [1], [37]

```
1:  INPUT:
2:      subsequence length m
3:      time series A of length n with elements A[1] ... A[n]
4:      time series B of length n with elements B[1] ... B[n]
5:      row and column offsets in distance matrix i, j
6:      dot product Q_{i-1,j-1} of subsequences A_{i-1,m} and B_{j-1,m}
7:      precomputed meta series μ̃_A, s (elements μ̃_A[1] ...)
8:      intermediate matrix profile and index P_AB, I_AB, P_BA, I_BA with elements P_AP[1] ...
9:  OUTPUT:
10:     Q_i,j                                  ▷ correlation score between A_{i,m} and B_{j,m}
11:     P_AB, I_AB, P_BA, I_BA                 ▷ updated matrix profiles and indices
12:
13: PROC:
14:                                            ▷ dot product of subsequences A_{i,m} and B_{j,m}
15: Q_{i,j} ← Q_{i-1,j-1} + A[i+m-1] · B[j+m-1] - A[i-1] · B[j-1]
16: k_{i,j} ← Q_{i,j} · (s_B[j] · s_A[i]) - μ̃_B[j] · μ̃_A[i]    ▷ correlation score of the subsequences
17: if (k_{i,j} > p[i]) then     ▷ new maximum score: update the horizontal result at location i
18:     P_AB[i] ← k_{i,j}
19:     I_AB[i] ← j
20: end if
21: if k_{i,j} > p[j] then       ▷ new maximum score: update the vertical result at location j
22:     P_BA[j] ← k_{i,j}
23:     I_BA[j] ← i
24: end if
```

Disregarding the index arithmetic, there are five floating point multiplications and 3 additions (including subtractions) in lines 14 and 16 of algorithm 1. Also the branching conditions in lines 17 and 21 need to be counted as FLOPs. For this reason the kernel is composed of a total of 10 floating point operations.

Additionally counting the number of memory accesses, we can estimate the arithmetic intensity of the kernel in order to reason about its possible performance on a machine. The exact number of memory accesses depends on whether the branching conditions are fulfilled and

whether the matrix profile is updated accordingly. Possibly $Q_{i,j}$ can be implemented as a register instead of being read from and written to memory, depending on the iteration scheme. The memory accesses are necessary for the STOMP iteration scheme, but not the SCRIMP one. As a simplification we assume, that either both update conditions are met and the profile updated at both locations i and j or that none of them is updated. This distinction suffices to provide the maximum and minimum number of memory access for both implementation cases of $Q_{i,j}$.

Table 1 Number of memory accesses of the kernel in algorithm 1, distinguishing whether the update conditions in lines 17 and 21 are met

	updates skipped	updates performed
$Q_{i,j}$ as register	10 float	12 float + 2 int
$Q_{i,j}$ in memory	12 float	14 float + 2 int

Assuming a implementation based on double precision floating point variables and $8\,\mathtt{byte}$ long integers, this yields arithmetic intensities between 0.08 FLOP/byte and 0.013 FLOP/byte as stated in Table 2. Those values are similar to the 0.08 FLOP/byte of the TRIAD kernel of the STREAM benchmark [47], which is used to measure the memory bandwidth of computers. This indicates, that the kernel is likely to be limited by memory bandwidth on most machines.

Table 2 Arithmetic intensities of the kernel in algorithm 1 based on double precision floats.

AI / FLOP/byte	updates skipped	updates performed
$Q_{i,j}$ as register	0.13	0.089
$Q_{i,j}$ in memory	0.1	0.078

4.2.2 Blocking Iteration Schemes

Implementations utilizing the optimized kernel of section 4.1 in the SCRIMP and STOMP iteration schemes, as reviewed in section 3.2.1, are typically bound by memory bandwidth due to the low arithmetic intensity of the kernel (see section 4.1). For that reason it is highly disadvantageous, that the SCRIMP (as well as the STOMP) scheme iterate several times linearly over large portions of the time series: in the most extreme case those portions span the whole length of the matrix profile. On machines with a hierarchical cache architecture the computational speed will be bound by the lowest memory hierarchy in which the time series can be kept. For huge problem sizes this will be the DRAM. This behavior was also empirically verified, as presented in the result section 6.1.

As an optimization on cached architectures, the cache locality can be improved by blocking

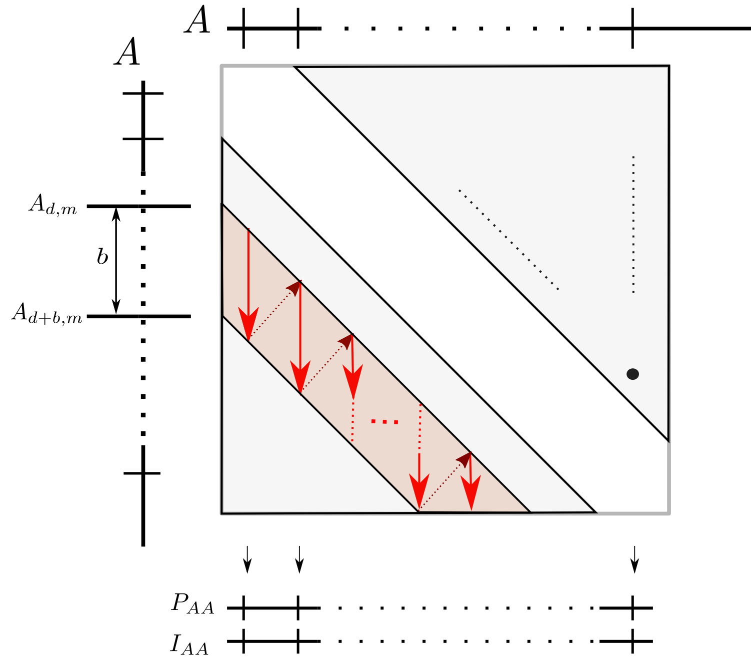


Figure 10 Vertical blocking scheme: A group of b neighboring diagonals (shaded in red) are evaluated during a single iteration over the time series. Iteration blocks are composed of small columns in the distance matrix. Their height is denoted as the block length b. A outer iteration over subsequent blocks, which start at a base diagonal d yields efficient evaluation of the group of diagonals. At the iteration end $b-1$ incomplete column blocks need to be considered

of kernel invocations with memory access to spatially close by memory locations. Properly modified iteration schemes could at least lift the performance to be bound by memory bandwidths of higher cache levels or even achieve full exploitation of the computational resources, depending on the exact peak performance and bandwidths. From the different possible block-wise iteration schemes, we selected one that also favors vectorization.

Vertical blocking

To achieve the blocking, we introduce an inner loop, iterating over small columns in the distance matrix. The outer loop covers all entries of the distance matrix, by applying SCRIMPs diagonal iteration scheme with such columns instead of single entries. Figure 10 illustrates the concept. Starting from a entry $d_{i,j}$ in the distance matrix, a block with block length b is composed of the entries $d_{i,j}, d_{i+1,j}, \dots d_{i+b,j}$. The values $d_{k+i,i}$ along a base diagonal in the distance matrix form the starting points for subsequent evaluations of blocks. Therefore the overall iteration scheme evaluates coherent groups of b neighboring diagonals. The scheme is formally stated in pseudo-code in Listing 2

The different blocks in the lower triangle are again evaluated in random order: the starting points of the blocks are distributed with a stride equal to b, with the very first block beginning

right after the exclusion zone.

Algorithm 2 Vertically blocked SCRIMP iteration scheme for a self-join on time series A

1: INPUT:
2: length of exclusion zone e
3: time series A of length n
4: subsequence window length w
5: block-length b
6: precomputed meta series $\widetilde{\mu}_A$, s_A
7: result buffers $\mathrm{P_{AA}}, \mathrm{I_{AA}}$
8: OUTPUT:
9: $\mathrm{P_{AA}}, \mathrm{I_{AA}}$ ▷ matrix profile and index
10:
11: PROC:
12: $\mathrm{profile_length} \leftarrow \mathrm{n} - \mathrm{w} + 1$
13: $\mathrm{start_rows} \leftarrow [\mathrm{e}, \mathrm{e}+\mathrm{b}, \mathrm{e}+2\mathrm{b}, \mathrm{e}+3\mathrm{b}, \ldots \max_{x\in\mathbb{N}}\{\mathrm{e}+x\cdot\mathrm{b} | \mathrm{e}+x\cdot\mathrm{b} < \mathrm{profile_length}\}]$
14: $\mathrm{base_diags} \leftarrow \texttt{shuffle}(\mathrm{start_rows})$
15: **for** $i \in \{e, \ldots \mathrm{profile_length}\}$ **do**
16: $\mathrm{tmp_Q}[i] \leftarrow \texttt{dotproduct}(A_{0,m}, A_{i,m})$ ▷ init dotprod. of each diagonal, equ. 4.2
17: **end for**
18:
19: **for** d in $\mathrm{base_diags}$ **do**
20: **for** $\mathrm{start_row} \in \{d, \ldots, d+\mathrm{b}-1\}$ **do**
21: $\mathrm{col} \leftarrow \mathrm{start_row}$
22: $\mathrm{row} \leftarrow \mathrm{start_row}$
23: **while** $\mathrm{row} < \min(\mathrm{start_row}+\mathrm{b}-1, \mathrm{profile_length})$ **do**
24: $\mathrm{tmp_Q}[\mathrm{row}] \leftarrow \texttt{eval_entry}(A, A, \mathrm{row}, \mathrm{col}, \mathrm{tmp_Q}[\mathrm{row}], \widetilde{\mu}_A, s_A, \mathrm{P_{AA}}, \mathrm{I_{AA}})$ ▷ see alg. 1
25: $\mathrm{row} \leftarrow \mathrm{row} + 1$
26: **end while**
27: **end for**
28: **end for**

One interesting property of the iteration scheme is, that all of the entries within a column block starting at $d_{k+x,x}$ affect elements p_x and i_x of the matrix profile and index. For this reason it is possible to hold the minimum distance value and the according index of a block within a register and only write the final result into memory. This way the arithmetic intensity of the kernel is improved, i.e increased: The memory accesses of the second update step in the kernel in line 21 of algorithm 1 are only necessary once per block instead of once per kernel evaluation.

In contrast to the original SCRIMP iteration, the results of the dot product $Q_{i,j}$ computed in

the kernel need to be stored in memory. Our choice was to create another meta-time-series tmp_Q with the same length as the matrix profile. It is initialized with the dot products of each diagonal before the iteration starts. As those are only needed once, during the iterations the values are replaced with the latest temporary intermediate result of individual diagonals.

Another desirable property of the iteration scheme is, that it favors SIMD vectorization, because the computations of the distance entries withing one block, i.e the kernel invocations in the innermost loop (line 23 in listing 2), are independent from each other.

The choice of the block length b is of tremendous importance. There are two limit cases: setting $b = 1$ results in the regular SCRIMP algorithm, as all diagonals are processed independently from each other in random order. Memory access is streaming over large fractions of the (meta) time series $n - m + 1$ times. The other extremal case is a block length b which exceeds the profile length: $b > n-m+1$. In that case the lower triangle of the distance matrix is evaluated column after column. This is identical to the STOMP iteration scheme. Again the memory accesses stream $n - m + 1$ times over the meta time series. In both cases, if the problem size exceeds the last levels cache size, all memory accesses of the kernel are accessing the main memory. Hence the operational intensity at DRAM level equals the kernels arithmetic intensity and the algorithm is prone to being bound by memory bandwidth.

In order for the blocking to improve performance, two conditions must be fulfilled: First, all data accessed during iteration over one block need to fit into the cache. Second, the operational intensity on DRAM level needs to be bigger than the ridge point of the machines roofline model. Investigating the access pattern of the blocking scheme allows us to derive an estimate of suitable block-lengths for a given cache size.

The memory access pattern is depicted in figure 11. The areas shaded dark illustrate the accesses of the iteration over one block, the lighter shaded and vertically shifted areas depict accesses of the block following afterwards. As indicated in the graphic, all meta time series are accesses at offsets $start_row$ to $start_row + b$. At offset col all except for tmp_Q are accessed. Further accesses exclusively to A occur at $col + w$ and $start_row + w$ to $start_row + w + b$. Table 3 accumulates the exact number of accesses, assuming the worst case branching behavior of the kernel, i.e. that every possible memory location is accessed. The stated byte count refers to a implementation using double precision floats. From the total count of $48+56\cdot b$ bytes one can derive a upper limit of the block length, given the cache size. E.g. for a typical level 1 cache size of 32 kB and under the assumption that all addresses can be mapped without conflicts, the maximum block length is 571 matrix entries.

The required lower bound for performance improvements is derived from the operational intensity at DRAM level. As derived in section 4.2.1 the kernel consists of 10 FLOP. As one block aggregates b kernel evaluations, the overall FLOP count is $10 \cdot b$. Assuming, that all the data of a single block iteration fit into the cache, main memory accesses are only required when

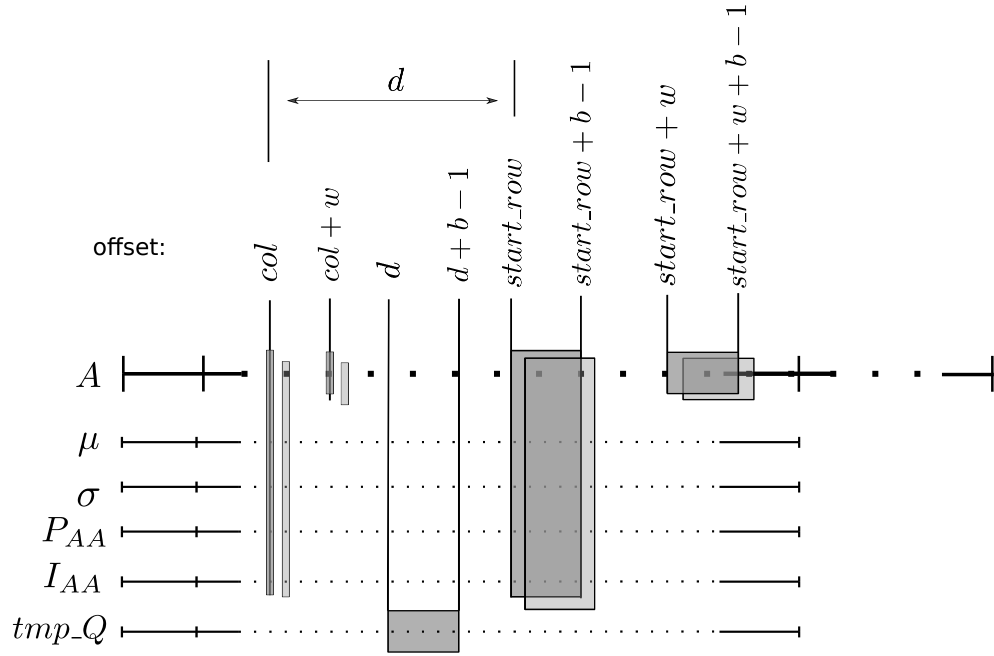


Figure 11 Access pattern of vertical blocking: The shaded areas indicate accessed variables in the data structures during evaluation of one block. The areas shifted slightly towards the bottom indicate the accesses in evaluation of the following up block. Thus the intersecting areas illustrate accesses which cane be served from cache in subsequent block evaluations, assuming that no conflicts occur and they do not exceed the cache size. The remaining areas need to loaded and or stored to main memory. Note that worst case behavior with respect to the branching in the kernel is assumed: every possible memory access is depicted.

switching from one block to the next one. In that case data not yet present in the cache needs to be loaded and previous block results need to be stored. For this reason the difference of the light and dark shaded areas in figure 11 gives an upper bound of their numbers: the difference in the shaded areas assumes write backs also for read only values like A. A more detailed analysis could give more exact counts but is not util here, as the writes constitute only a minor contribution and using a upper bound is sufficient to estimate minimum operational intensity. As there is a shift of exactly $1\,\mathtt{element}$ between the rectangles, the number of DRAM accesses is bound by twice the result of table 3 with $b = 1$. Assuming again a double precision implementation, the amount of DRAM transfers per block is $2 \cdot (48 + 56)\mathtt{byte} = 208\,\mathtt{byte}$. This gives an operational intensity of $(b \cdot 10/208)$ FLOP/byte at the DRAM level. Choosing b such that it exceeds the machine balance causes the algorithm to be compute bound. For our target machine, the DRAM ridge point is $0.16\,\mathrm{FLOP}\,\mathtt{byte}^{-1}$ (see figure 23). Gains form the blocking scheme can therefore be expected with block lengths $b \geq 4$, which is distant form the upper limit due to cache size and promises to achieve efficient cache use with this blocking scheme.

Table 3 Memory accesses in a iteration over a vertical block. Numbers are computed from accesses as depicted in figure 11. Branching in the kernel is neglected, i.e. it is assumed that every memory location to be accessed. The number of bytes assumes 8 byte floats and 8 byte integers as in a x86 implementation with double precision floats

starting offset of access-block	floats	ints	bytes
col	4	1	40
$col + w$	1	0	8
d	1 $\cdot b$	0	8 $\cdot b$
$start_row$	4$\cdot b$	1 $\cdot b$	40 $\cdot b$
$start_row + w$	1 $\cdot b$	0	8 $\cdot b$
all summed up	5+6$\cdot b$	1+1$\cdot b$	48 + 56$\cdot b$

4.3 Vectorization

In order to achieve highest hardware utilization, we tried to achieve SIMD vectorization. Subsequent iterations of the innermost loop of the vertically blocked iteration scheme (23 in algorithm 2) exhibit independence in computation of the sliding dot products, correlation scores (lines 14 and 16 in algorithm 1) as well as updating the vertical profile and index (lines 21 to 24). In contrast to that, update of the horizontal result (lines 17 to 20) exhibits a loop-carried dependency between the read, i.e. between the read in the update condition (l. 17) and the potential write of a new result in the previous iteration (l. 22). Still instruction level parallelism can be achieved by proper treatment: the update resembles a reduction among all the loop results. Update of the profile P_{BA} constitutes a plain maximum reduction. Update of the index I_{BA} has the same structure as a maximum reduction, with the particularity, that updated value is different from the determining maximum.

For one specific kernel implementation of a AA self-join kernel, we were able to make use of auto-vectorization with Intels C++ compiler but not the general AB-join kernel, which we required in a parallelization. While the specification of OpenMP allows implementation of custom SIMD reductions since version 4.0 [64, pp. 175, 188] our Intel compiler did not support it. We chose to perform vectorization manually, using compiler intrinsics for AVX2, FMA and the preceding SIMD instruction sets [65], [66]. As the column index j is constant within the loop over a column block we choose to store and update private copies of the result $P_{BA}[j]$ and $I_{BA}[j]$ for each vector lane. At the end of the vectorized loop, a reduction among the vector lanes is performed and the result written back to the memory locations.

4.4 Comparison to SCAMP

The SCAMP algorithm [2], which was published concurrently to our work, also applies optimizations to the kernel, with some similarities and some differences to ours, as detailed in this section.

According to the paper the algorithm builds on the STOMP scheme. For this reason in contrast to our work the kernel is stated for evaluation of the upper triangles, which due to the explained symmetry makes no difference. The STOMP iteration scheme was modified, such the the matrix is not processed one full row after another, but the triangle is split into parallelograms which are processed row by row. Taking the symmetry into account, this coincides with our blocked iteration scheme. While the publication focuses on the GPU implementation, the bandwidth limitation was stated there and the CPU code[1] shows the choice of a 256 sample block-length.

Furthermore the arithmetic was modified. The choices exhibit some similarities and differences to our work. All respective equations of both versions are shown next to each other for a comparison. Note that the precomputations are shown for mathematical completeness only but do not impact performance.

OURS

precomputations:

$$\widetilde{\mu}_i = \mu_i\sqrt{m}/\sigma_i$$
$$s_i = 1/\sigma_i$$

objective:

$$\widetilde{p_i} = \max_j k_{i,j}$$
$$p_i = \sqrt{2m - 2\widetilde{p_i}}$$

kernel

$$\begin{aligned} Q_{i,j} &= Q_{i-1,j-1} - T_{i-1} \cdot T_{j-1} \\ &\quad + T_{i+m-1} \cdot T_{j+m-1} \\ k_{i,j} &= Q_{i,j} \cdot (s_j \cdot s_i) - \widetilde{\mu}_j \cdot \widetilde{\mu}_i \end{aligned}$$

SCAMP

precomputations:

$$\mathrm{df}_i = 0.5 \cdot (T_{i+m} - T_i)$$
$$\mathrm{dg}_i = (T_{i+m} - \mu_{i+1,m}) + (T_i - \mu_{i,m})$$
$$a_i = 1/\|T_{i,m}\|$$

objective:

$$\overline{p_i} = \max_i \mathrm{CC}_{i,j}$$
$$p_i = \sqrt{2m(1 - \overline{p_i})}$$

kernel

$$\overline{QT}_{i,j} = \overline{QT}_{i,j} + \mathrm{df}_i \cdot \mathrm{dg}_j + \mathrm{df}_j \cdot \mathrm{dg}_i$$

$$\mathrm{CC}_{i,j} = \overline{QT}_{i-1,j-1} \cdot a_i \cdot a_j$$

Similar to ours, the SCAMP kernel avoids costly operations like divisions in the kernel by moving them to the precomputations. In both versions, the objective was turned into a maximization objective in order to avoid unnecessary computations. The euclidean matrix profile

[1] available at https://github.com/kavj/matrixProfile at the time of our work

is reconstructed in a post-processing step. SCAMP maximizes the Pearson correlation coefficient CC, our version the m-th fraction of it. The multiplication could be subsumed in SCAMPS precomputations and has no impact on the kernel performance.

The SCAMP kernel exhibits a rather different computation scheme though. The employed *centered-sum-of-products* formula for computation of the dot products is even more efficient: only 6 FLOP are required, compared to 8 FLOP in our version. At the same time also the number of variable accesses had been reduced by two. Taking into account the additional 2 FLOP and worst case 4 memory accesses for the required profile updates, the expected arithmetic intensity of the kernel is 0.083 FLOP/byte (assuming that none of the variables can be buffered in a register between kernel iterations). As this AI is almost the same as for our kernel, it is to be assumed, that the kernels can achieve the same peak compute rate on a machine. Because the SCAMP kernel requires less FLOP for evaluations of a matrix entry it promises a higher throughput than ours. Namely a speedup of $(8+2)\text{FLOP}/(6+2)\text{FLOP} \approx 1.25$ compared to our kernel is to be expected based on these intensities. As we already finished our sequential optimizations at the time of the publication and due to the time limit we did not incorporate their improvements into our work.

5. Parallelization

In the original SCRIMP publication [32] it is outlined, that a straightforward way to parallelize the computation can be achieved by distributing the independent evaluation of diagonals in the distance matrix. We examine the approach in depth, applying the previously discussed sequential optimizations and some modifications for improved load balancing. The theory and implementation is explained in section 5.1, results of the conducted experiments are presented in 6.2.

As this trivial parallelization exhibits weaknesses like a limitation of the maximum problem size, we propose our own solution in section 5.2. It allows processing of arbitrarily large problems and also reduces the communication overhead.

5.1 Trivial Parallelization

Zhu *et al.* [32] outlined, that diagonals of the distance matrix in the SCRIMP algorithm can be evaluated independently. The publication hinted at the immanent parallelization resulting from that independence. To examine the behavior, we implemented this approach with minor customization, restricting ourselves to the scenario of self-similarity search. After a outline of the approach, we generically discuss the parallelization strategy. Details of the implementation and the mapping to MPI are elaborated in section 5.1.2

5.1.1 Algorithm

The basic parallelization approach is, that each process independently evaluates a subset of the diagonals. In this way each process is producing a intermediate local version of the matrix profile. Those partial results are communicated and merged in order to produce the final global result. Listing 5.1 lists all steps of the algorithm, which will be investigated in more detail in the following subsections.

A immediate observation in listing 5.1 is, that the only part of the algorithm which requires communication among workers is merging of the local results in line 12. Further parallel overhead is created by the redundant loading of the time series in line 2 up to the precomputations in line 4. Actually the precomputations could be performed in parallel by splitting along time and the results communicated. Because intermediate results of the sequential analysis suggested that little gains are to be expected, as the precomputations require very little computation time, we omitted distributing the precomputations (see e.g. figure 27a). A higher impact has loading and distributing the input data as well as storing the result, as this inherently requires network communication and is constrained by the performance of the storage system. For that reason, we tried to take advantage of parallel I/O capabilities.

```
1    for each worker with ID wid do in parallel:
2        A = load_time_series()
3        localP_wid, localI_wid = allocate_result_buffers()
4        μ̃, s = precompute_meta_series()   // as in section 4.1
5        local_partition = obtain_local_set_of_diagonals()
6        for each diagonal in local_partition do
7            tmp_Qs[index of diagonal] = compute_initial_dotproduct(diagonal)
8        local_blocks = decompose_into_blocks(local_partition)
9        for block in local_blocks do
10           evaluate_block_of_diagonals(block, tmp_Qs, localP_wid, localI_wid) // as in
                 algorithm 2
11       transform_scores_to_distances(localP_wid) // as in equ. 4.8
12   P, I = merge_local_results( { (localP_1, localI_1), (localP_2, localI_2), ...} )
13   store_result(P, I)
```

Listing 5.1: Straightforward parallelization of the SCRIMP scheme

Partitioning

We applied a partitioning scheme for line 6 in listing 5.1, which manually balances the work-load among the processors. The approach is illustrated in figure 12 for the case of three workers by painting the partition of each worker in one color: As we are concerned with the symmetric distance matrix of the self-similarity search, only the lower triangle of the distance matrix (sans the trivial matching zone) along the main diagonal, needs to be distributed among the workers. Obviously the diagonals in that section vary in length: the diagonals closest to the main diagonal are longest, hence most work intensive, while the ones towards the edge of the matrix are shorter. To balance the work, the work partition of a processes is composed of two chunks: Let p denote the given number of workers, which is chosen as $p = 3$ in illustration12. In order to balance the load among the workers, we divide the legs of the lower triangle of interest into $2 \cdot p$ parts. For simplicity at this point it shall be assumed, that all of them are of equal size. Connecting the corresponding endpoints of those sections in the last row and first column of the distance matrix defines slices of the work triangle. The partition which is to be processed by worker $x \in 1, ..., p$, is composed of the slices x and $p - x + 1$, which reside symmetrically in the triangle. Due to the symmetry, all the partitions, composed of two symmetric slices, contain the same amount of work, which is defined by its area. Overall every worker has to process a number of diagonals, which is the p-th fraction of the total number of diagonals. This implies, that the initialization work, namely computing the dot product Q for the very first entry of a diagonal, is also equally distributed among the workers.

Merging the individual results

As soon as all the processes finished their local evaluations, every worker with id wid holds a intermediate matrix profile and profile index $\mathrm{localP_{wid}}$ and $\mathrm{localI_{wid}}$. To obtain the global

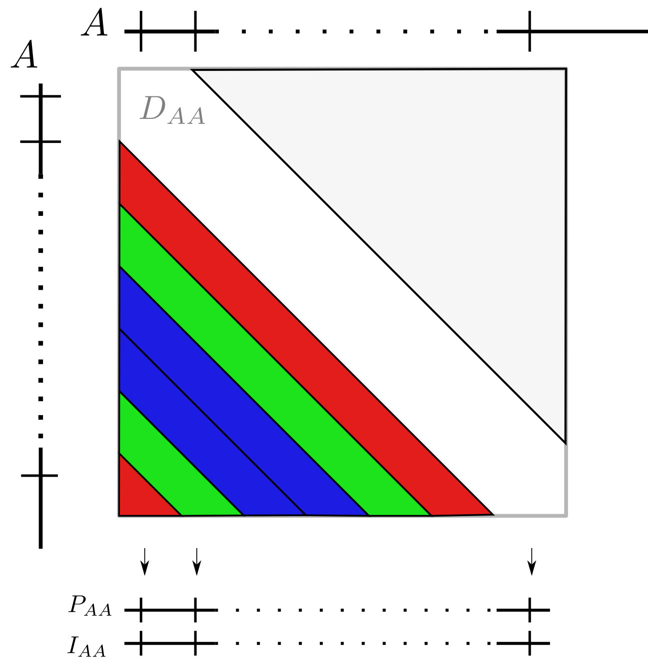


Figure 12 Partitioning of work among processes. The illustration shows partitioning for three processes: each color corresponds to one process and illustrates the part of the distance matrix, over which it has to iterate. Again symmetry of the self-join is exploited to omit iteration over the upper part of the distance matrix. Note how each worker has to process two partitions in the lower triangle to process in order to achieve load balancing.

result, they need to be merged: for each time point, the valid index and profile value pair is the one with the lowest profile value. Algorithm 3 demonstrates, how to merge two such local results. The global result of all intermediate local profiles is obtained by successive pairwise merging.

As outlined in listing 5.1, the merged result composes the global result of the algorithm and needs to be written to a file afterwards. It depends on the file output implementation, which workers are required to obtain the final result. Three potential choices are available:

1. Only a master process, solely responsible for the file output, receives the result
2. All processes receive the global result and can participate in file writing
3. A subset of workers receive specific sections of the result, which they will write to a file

In order to take advantage of the parallel file system capabilities at our system (see section 8.1) we chose to implement variant 2 and spread the result to all workers. This variant can also be directly mapped to MPI routines, in contrast to variant 3, which could possible provide lower I/O overheads, if a optimum choice for the number of output processes is made. As we consider the implementation only a intermediate step and baseline, we omit the additional implementation and benchmarking effort for such a scheme.

Algorithm 3 Merging two local matrix profile results

```
1: INPUT:
2:     profile length l
3:     first intermediate matrix profile and index (P_1, I_1)
4:     second intermediate matrix profile and index (P_2, I_2)
5: OUTPUT:
6:     merged profile and index (P_out, I_out)
7:
8: PROC:
9: for all i ∈ 1, ..., l do
10:     if P_1[l] < P_2[l] then
11:         P_out[l] ← P_1[l]
12:         I_out[l] ← I_1[l]
13:     else
14:         P_out[l] ← P_2[l]
15:         I_out[l] ← I_2[l]
16:     end if
17: end for
```

File I/O

Because to the previously outlined partitioning, every worker requires all the time series data as input: the partition of a arbitrary process contains distance matrix entries of the very first column as well as the last row, as can be seen in figure 12. Computation of the first entries requires the very first sample value in the time series, computation of the latter ones involve the samples from the end.

In order to keep all processes busy and take advantage of the parallel file system, we partition the input workload among all processes. All the chunks need to be distributed afterwards to all processes, such that in the end everyone holds the complete time series.

We partition the one-dimensional time series of length N into p chunks of roughly equal size, where p denotes the number of processes. Length and offset of chunk $i \in \{0, ..., p-1\}$ are computed according to equation 5.1

$$
\begin{aligned}
\text{chunk length}(i) &= \begin{cases} \lfloor N/p \rfloor & \text{iff} \quad i < N \bmod p \\ \lceil N/p \rceil & \text{else} \end{cases} \\
\text{chunk offset}(i) &= \begin{cases} i \cdot \lfloor N/p \rfloor & \text{iff} \quad i < N \bmod p \\ i \cdot \lfloor N/p \rfloor + i - (N \bmod p) & \text{else} \end{cases}
\end{aligned}
\tag{5.1}
$$

After reading, a all-to-all exchange of the chunks provides each process with the full series.

Reading in parallel in this way is expected to be most efficient, when taking advantage of parallel file system features, which are potentially available at the HPC system. Utilization of such can be achieved by utilization of the MPI I/O interface. As the interface provides binary file access, we employed a binary input file format for the time series. It consists of a small header block and the sequentially written time series samples. As the details are MPI related, we defer them to section 5.1.2. The specification of the series length in the header is necessary to compute the chunk sizes and offsets without reading all the input data. We chose to read the header from every process, hoping that potential caching mechanisms at the storage nodes in the MPI implementation yield a good performance.

It is to be be noted that it is uncertain, whether it is actually the best choice, to perform parallel I/O from all processes: depending on the library internals, communication overhead between all participating processes is involved. Typically the number of storage nodes is limited and network performance is shared, i.e. between the processes residing on the same node. Using only a subset of prcessis could potentially show lower overheads and further work could go into investigations of the optimal configuration of input processes.

For file output we apply the same scheme: the matrix profile if length N is partitioned into p chunks as in equation 5.1. All p processes write once such contiguous section with a binary file format as specified in the MPI details.

Limitations of the approach

As pointed out in the discussion of file input, the presented partitioning requires every worker to load the full input time series. Furthermore also the full length matrix profile needs to be kept in memory of every single process. For this reason the limited amount of memory at a single node implies a upper bound for the length of the matrix profile computed with this trivial parallelization.

Let size_float denote the number of bytes occupied by the floating point type used for representation of the time series samples as well as the profiles distance values and meta-time series, as presented in 4.2.2. Let size_int denote the byte-size of the profile-indexes integer datatype, M the maximum amount of memory at a node and p the number of processes per node. There are 3 floating point meta time series required in addition to the time series and profile distance, as depicted in 11. For this reason a upper bound $N_{\max}$ for the maximally feasible time series length can be obtained by assuming a $(2+3) \cdot \text{size_float} + \text{size_int}$ bytes memory requirement for each time series sample value:

$$n_{\max} = \frac{M}{(\text{size_float} \cdot 5 + \text{size_int}) \cdot p} \tag{5.2}$$

For the hardware of our experimental system, the supermuc phase 2 whose specifications are listed in 8.1, we obtain for example the following maximum input length, under the assumption

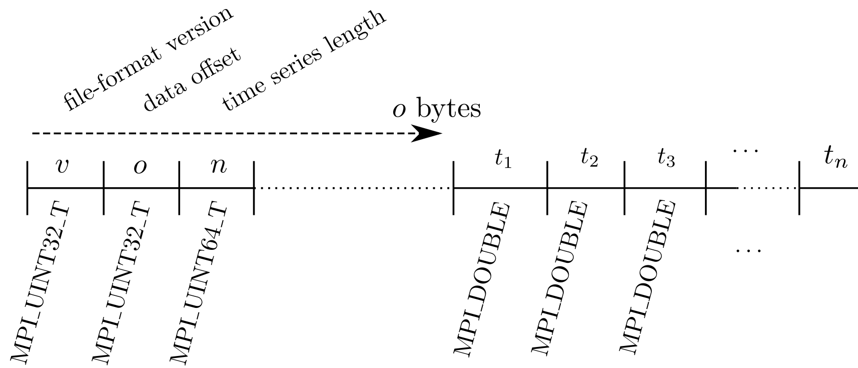


Figure 13 Binary time series file format: The file starts with a header at byte 0: a 32 bit unsigned int specifies a file format version, another 32 bit unsigned int designates the byte offset of the first time series sample from the start of the file. Furthermore the header contains a 64 bit unsigned integer for specification of the time series length. The actual sample values are sequentially written as MPI_DOUBLE values, starting at offset o as specified in the header.

of 8 byte double precision floats and 4 byte integers:

$$n_{\text{max, supermuc phase2}} = \frac{64 \cdot 10^9 \text{ byte}}{44 \text{ byte} \cdot 28} \approx 52 \cdot 10^6 \tag{5.3}$$

As full usage of the available memory is unrealistic, further data, like MPI internal buffers as well as operating system data are not taken into account, the implementation will be restricted to even smaller time series lengths.

5.1.2 Mapping to the MPI

Time series file input

For parallel reading of the the time series, we employed a binary input file format, motivated by the fact that the MPI provides a binary file access interface. We provide a simple conversion utility, which enables conversion from ASCII files to our format. Our file format consists of two sections and is illustrated in figure 13: It starts with a header section, containing a file-format version, the byte offset of the actual time series data within the file and the length of the time series. The time series samples are written as a sequence of MPI_DOUBLE values, starting at the offset specified in the header.

Parallel reading of the data chunks is implemented as a blocking collective MPI_File_read_all through a file view. The file view maps only the data chunk within the file which is relevant to the respective process. It is created with MPI_Type_create_subarray as a slice of a one-dimensional array of MPI_DOUBLE. In order to determine the starting offset and length of the chunk, every process reads the header structure as a derived MPI datatype through a individual MPI_MPI_File_read_at before reading the actual data portion.

Input communication

After distributed reading of the input time series from the binary file, as explained in the previous section, all read chunks need to be distributed to all processes. By reading the data chunks into the respective sub-array of a global time series input buffer, a call to MPI_Allgatherv can be used afterwards to spread the full input series to all processes.

Result reduction

The only step in this trivial parallelization, as previously outlined in listing 5.1, which necessarily requires communication, is merging the processes partial results (line 12 in the pseudocode). Within the MPI library, it can be naturally expressed as a reduction operation based on the pairwise merging procedure in algorithm 3. Using MPI_Allreduce, the full result becomes available to all processes. While every process requires only a specific chunk for the outline parallel I/O operation, for simplicity of implementation we rely on MPI_Allreduce.

As the MPI reductions are designed as binary operators, it is necessary to express the pair of matrix profile and its accompanying index as a MPI Datatype. In order to avoid redundant memory allocations and copying of the data, we create a derived MPI datatype such that it maps to the structure of arrays, which is is used for kernel computation.

The data length parameter of the reduction operation is equivalent to construction of a contiguous datatype [67, section 4.1.11] which concatenates elements based on their extent [67, pp. 85 sq.]. Due to this fact it is impossible to model the structure of arrays as a array of overlapping (profile, index)-pairs. For this reason we construct our derived MPI datatype as single structure of two contiguous blocks, one for the array of profile indices and one for the similarity value. The block length is set equal to the result length, which is dynamically determined according to the input time series length and invocation arguments. With such a datatype, the reduction operation is always invoked with a length of $1\,\text{element}$. Complementary to this choice, the user defined reduction operation dynamically decodes the result length from the derived MPI_Datatype.

Result writing

Writing with MPIs parallel I/O capabilities is implemented with a custom binary file format, very similar to the previously explained binary input format. It consists of a small header and the data section, as illustrated in figure 14. The matrix profile is stored as an array of structures. All processes participate in parallel writing and the 1-dimensional matrix profile index space is partitioned as in equation 5.1. While the header is written only by the master with individual write, the data is written with the blocking collective MPI_File_write_all. A derived datatype for the distance and index pair (p_x, i_x) allows access through a file view created by

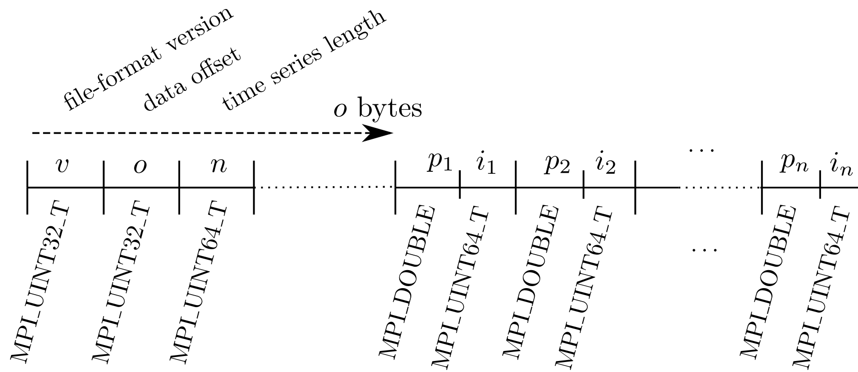


Figure 14 Binary matrix profile file format: The file starts with a header at byte 0, stating a file format version, the byte offset of the data section from the start of the file and the number of stored matrix profile entries. The profile values itself are stored as a array of structures, starting at offset o as specified in the header. Each entry consists of the distance value p_x and the profile index i_x.

MPI_Type_create_subarray. As the partitions are not overlapping, we hint the implementation at exlusive writing with MPI_File_set_info.

5.1.3 Theoretical Performance Analysis

In order to provide a analysis of the theoretical scaling behavior of the implementation and its isoefficiency metric, we ignore the impact the window length w and exclusion zone e. It simplifies the model and they are supposed to be very small compared to the total time series length [1]. With this simplification, we assume the matrix profile length to be equal to the input series length n.

Table 4 lists models of the runtime behavior for the most important program sections (compare results section 6.2). The analysis considers the cost, i.e the accumulated time over all processes.

Under our simplifications, the distance matrix contains n^2 entries, which is equal to the number of kernel evaluations. Because the kernel contains a fixed amount of arithmetic operations the arithmetic work W is proportional to the number of kernel invocations, i.e. $W \propto n^2$. While some of the precomputations are also required for the overall computation, we do not consider them in the definition of our work amount, to keep the analysis simple. It can be seen in our experimental results (sec. 6.2), that dot product initialization and precomputation times are neglectably low. The right column in table 4 list the respective scaling behaviors as a function of W.

As the implementation distributes full diagonals among the processes, only one dot product initialization per diagonal is required. As a result the respective accumulate time is independent of p and proportional to the time series length n. For this reason the respective parallel time is growing with $n/p = \sqrt{W}/p$, which is slower than the kernel cost. This will show up as a additional speedup in the experiments (sec. 6.2). In contrast to it, the precomputations are

Table 4 Theoretical scaling analysis of trivial parallelization: simplified scaling behavior of important program sections. The middle column states the approximate runtime behavior depending on the number of processes p and the input length n. The rightmost column lists the same behavior based on the problem size $W \propto n^2$. *Total runtime* refers to the accumulate time over all processes spent in distinct code sections. For I/O bandwidth and latency bound behavior is stated, agnostic of any MPI internal behavior

functionality	**total runtime** $T(p,n)$	**total runtime** $T(p,W)$
precomputations	$\Theta(p \cdot n)$	$\Theta(p \cdot \sqrt{W})$
dot products	$\Theta(n)$	$\Theta(\sqrt{W})$
kernel evaluations	$\Theta(n^2/2)$	$\Theta(W)$
result reduction	$\Theta(p\log(p) \cdot n)$	$\Theta(p\log(p)\sqrt{W})$
File I/O	BW: $\Omega(p \cdot n/p)$ Lat: $\Omega(p)$	BW: $\Omega(\sqrt{W})$ Lat: $\Omega(p)$

performed redundantly by every process, yielding $\Theta(p \cdot n)$ behavior.

Behavior of any communication is not only depending on our implementation but also on the underlying MPI implementation, file system and communication network. Theoretical performance analysis in the literature often refers to a *parallel system* rather than a implementation. In particular MPI internal communication is to be suspected regarding the file I/O routines [68].

Because we found no reliable information for our target system, the stated I/O behavior in table 4 is based only on our implementations properties and therefore constitutes only a lower bound: in particular all p processes are writing or reading slices of length n/p. As long as the bandwidth is dominating, this yields a total runtime behavior of $\Omega(n)$. With increasing numbers of processes, the output slices shrink and a latency-bound behavior of $\Omega(p)$ might be observed.

We assumed the result communication to be dominated purely by bandwidth. This is reasonable, as always the full n-entry matrix profile is reduced. Under the assumption of an optimized tree-style reduction agnostic of the network with a $log(p)$ parallel runtime, the resulting total runtime behavior is $\Theta(p\log(p) \cdot n)$.

Based on the stated idealized scaling behaviors, it is to be expected that in any scaling scenario the long-term dominating trend is caused by the result communication overhead.

Using additional constants, the listed runtimes can be composed into a performance model for the total processing time:

$$T_{total,triv} = c_1 + c_2 \cdot W + c_3 \cdot p \cdot \sqrt{W} + c_4 \cdot p\log(p)\sqrt{W} + c_5 \cdot \sqrt{W} + [c_6 \cdot p] \tag{5.4}$$

The isoefficiency metric [62] in our case it is obtained by balancing the workload against the result reductions (see sec: 3.3.2):

$$\begin{aligned} W &= K\sqrt{W} \cdot p\log(p) \\ \implies W &= K^2 p^2 \log^2(p) \end{aligned}$$

The isoefficiency metric of the trivial parallelization is given by $\Theta(p^2 \log^2(p))$. According to it, the input length n needs to be increased with at least $p\log(p)$ for cost-optimal scaling.

5.2 Distributed parallelization

As shown in section 5.1.1, the trivial parallelization approach requires to hold the whole input series in memory and for this reason the available memory of a single-node limits its scalability to long or high-detailed time series. Motivated by this fact we present a parallelization approach, which loads only fractions of the overall series into the memory of each processes. Furthermore parallel overheads are reduced with the presented approach, which will be verified in experiments in section 6.4.1.

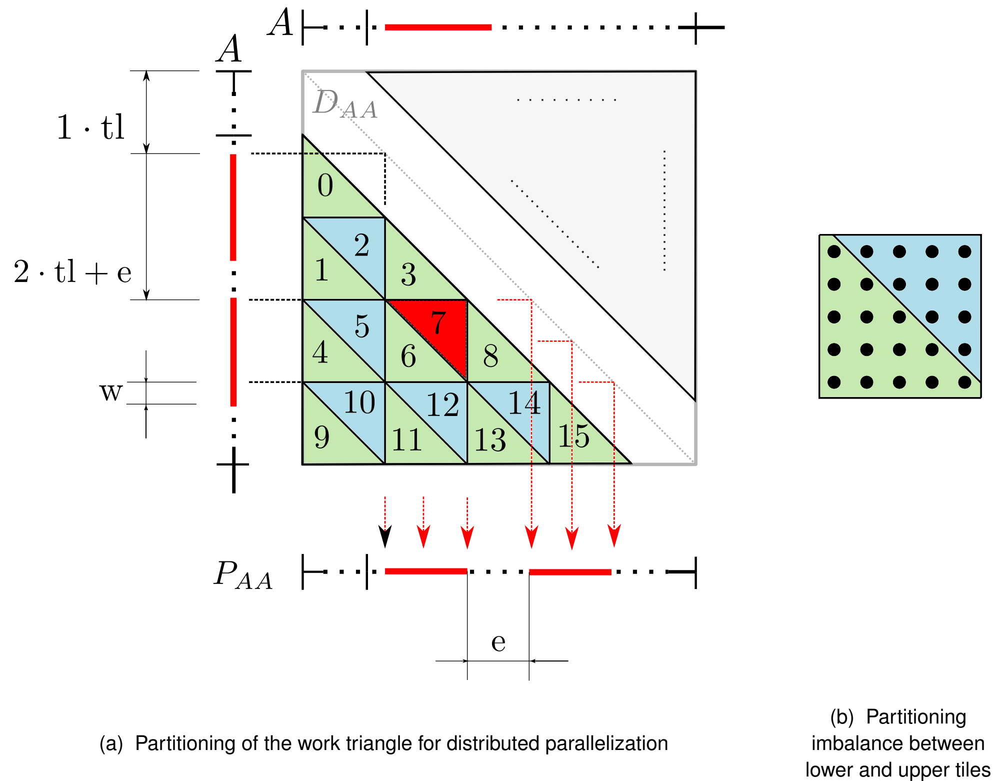
We start with a general presentation of the approach in section 5.2.1, before outlining the implementation based on MPI 5.2.2. We conclude our theoretical discussion with a comparison to the SCAMP framework [2], which was published concurrent to our work.

5.2.1 Algorithm

The problem size limitation of the trivial parallelization scheme (sec. 5.1.1) can be overcome by a modified partitioning scheme. Splitting the working triangle into smaller triangular tiles, as in the SCAMP publication [2], requires each process to hold only two contiguous subsections of the overall time series and matrix profile. Figure 15a gives an overview of the partitioning and will be explained in detail later on. Our implementation uses a one-to-one mapping of tiles to processes and therefore requires that the available number of processes is always square. This approach is similar to the checkerboard based matrix-vector product discussed by Grama *et al.* [62]. It has the advantage of a comparatively small communication overhead, at the cost of being restricted a to square number of workers.

Algorithm 4 outlines the parallelization: A subset of the processes reads in slices of the time series, such that the complete time series data is distributed across the cluster. Workers, which did not participate in reading, receive their required input subsequences from the input processes. Afterwards, every process independently performs the necessary precomputations and evaluates all matrix entries in its local tile. The local result of the evaluation is two partial result slices of the global matrix profile: one slice, called *horizontal*, corresponds to the profile of the local tile. The second *vertical* slice, corresponds to the profile produced by the symmetric section in the upper triangle. It contains the profile values generated when looking for the row-wise minima in the symmetric kernel (algorithm 1). The result slices are coarsely

aligned according to the row or respectively column coordinates of the tile. As several processes contribute to each slice of the global profile, partial results contributing to the same slice need to be merged. After the merging procedure, each slice is aggregated in a specific output process, which will write the result to a file. We will discuss the details of the individual steps in the following sections, starting with the partitioning of the workload and data, as it motivates the other implementation choices.



(a) Partitioning of the work triangle for distributed parallelization

(b) Partitioning imbalance between lower and upper tiles

Figure 15 Partitioning for distributed parallelization: subfigure (a) depicts partitioning for $p = 16$ processes. The work triangle in the lower half of the distance matrix is split into p small triangular tiles, arranged in $\sqrt{p} = 4$ rows and columns. Each process is responsible for evaluating one such tile. Generally, the computation of a tile requires two sub-series of the input and produces a partial result contributing to two subsections of the the final matrix profile, which are illustrated for the tile highlighted in red in the second column and third row. Starting points of the input/output sections correspond to horizontal/vertical projections of a tile onto the input/output series. The exclusion zone of length e (see 3.1.1) causes a offset between starting points of the subsections: denoting the tile length as tl, the "horizontal" input section of the highlighted tile starts at $(2-1) \cdot \mathrm{tl}$, as the tile is in the second column. The "vertical" input section though is starting at $e + (3-1) \cdot \mathrm{tl}$. Additionally, the required input chunks' length exceeds the tile length by the user specified window length w.
Sub-figure (a) illustrates the partitioning details: dots represent entries of the distance matrix. As an example the partitioning of a square 5×5 matrix section into a lower and upper tile is shown. Evaluation of the distance entries along the diagonal is assigned to the lower triangle, causing an imbalance in the workload of the triangles

For all subsequent explanations of the parallelization scheme, let $n \in \mathbb{N}$ denote the length of input time series A. Let $w \in \mathbb{N}$ denote the subsequence window length and $e \in \mathbb{N}$ the length of the exclusion zone, whose semantics and restrictions are explained in section 3.1.1.

Algorithm 4 Distributed parallelization of the SCRIMP computation

1: INPUT:
2: square number of processes p
3: file with input time series T
4: window length winlen
5: OUTPUT:
6: output file with matrix profile $R = (P, I)$
7:
8: PROC:
9: **for** each worker with $\mathrm{wid} \in 0, \ldots, p-1$ **do in parallel**
10: $(\mathrm{tile_row}, \mathrm{tile_col}) \leftarrow$ retrive_partition_coords (wid)
11: **if** $\mathrm{wid} \in \{$input process ids$\}$ **then**
12: $(\mathrm{T_{vert_slice}}, \mathrm{T_{hor_slice}}) \leftarrow$ read_input_slices$()$ ▷ for assignment of slices see fig. 18c
13: spread_ts_slices$(\mathrm{T_{vert_slice}}, \mathrm{T_{hor_slice}})$
14: **else**
15: receive_ts_slices$()$
16: **end if**
17: $(\widetilde{\mu}_{\mathrm{vert}},\ \mathrm{s_{vert}},\ \widetilde{\mu}_{\mathrm{hor}},\ \mathrm{s_{hor}}) \leftarrow$ precompute_meta_series$(\mathrm{T_{vert_slice}},\ \mathrm{T_{hor_slice}}, \mathrm{winlen})$
18: $\mathrm{tmp_Qs} \leftarrow$ compute_initial_dotproducts$(\mathrm{T_{vert_slice}}, \mathrm{T_{hor_slice,0:winlen}})$
19: apply_input_padding$(\mathrm{tmp_Qs}, \mathrm{T_{vert_slice}}, \mathrm{T_{hor_slice}})$ ▷ necessary to enable partitioning
20: $(\mathrm{partial_R_slice_{tile_row,wid}}, \mathrm{partial_R_slice_{tile_col,wid}}) \leftarrow$ eval_local_tile_blocked$(\ldots)$
21: **end parallel for**
22: **for** each result slice $\mathrm{R_slice_i}$ with $i \in 0, \ldots, \sqrt{p}$ **do**
23: $\mathrm{R_slice_i} \leftarrow$ merge_partial_results$(\{\mathrm{partial_R_slice_{i,\ldots}}\})$ ▷ such that $\mathrm{R_slice_i}$ is finally available at the output processes responsible for writing
24: **end for**
25: **for** each worker with $wid \in \{$output process ids$\}$ **do in parallel**
26: transform_scores_to_distances$(\mathrm{R_slice_x})$ ▷ $\mathrm{R_slice_x}$ denoting the previously received slice
27: write_result_slice $(\mathrm{R_slice_x})$ ▷ the work assignment will be explained later
28: **end parallel for**

Partitioning

The workload of evaluating the lower triangle of the distance matrix, sans the exclusion zone, is split into smaller tiles, each a right triangle itself. The scheme is illustrated in figure 15a for 16 processes. Let $p \in \{x^2 | x \in \mathbb{N}\}$ denote the number of processes. The work triangle is partitioned into $\sqrt{p}$ rows and columns. Numbering of rows, columns and tiles is performed left to right and top to bottom, starting from 0. In this scheme, the leftmost tile in row $i \in 0, \ldots, \sqrt{p}$ has index i^2 and the rightmost tile index $(i + 1)^2 - 1$. Obviously, the partitioning requires tiles that are lower and upper triangles which are shaded in the illustration in green and blue respectively.

Similar to the work triangle, the tiles form isosceles right triangles, because the SCRIMP kernel iteration proceeds in a diagonal direction. As the matrix is composed of a discrete number of entries, the high-level view is a simplification. The partition pattern requires square sub-matrices to be decomposed into exactly two tiles. Figure 15b depicts our decomposition for the example of a 5×5 sub-matrix: in order to avoid redundant computations, we assign the evaluation of the squares diagonal to the lower tile. This implies that the top left corner of the upper triangle has an additional column offset of 1 compared to the lower tile, while both lie in the same row of the matrix. Thus, the width of the upper triangle is 1 matrix entry smaller compared to the lower one. We call the length of the lower triangle basis the *tile length*. It is identical to the base length of sub-matrices formed by pairs of lower and upper tiles. Compared to the lower tiles, the upper ones lack one column of $tile_length$ entries. As the lower tiles contain work proportional to $tile_length^2$, the resulting imbalance vanishes with increasingly large tiles, as it is proportional to $tile_length/tile_length^2 = 1/tile_length$.

The discrete nature of the distance matrix furthermore imposes a restriction on the size of the working triangle, such that the decomposition into equally sized right triangles is possible. For further explanation of the partitioning details, we will assume that the following condition holds:

$$\exists \text{ tile_length} \in \mathbb{N} : \text{tile_length} \cdot \sqrt{p} = n - w + 1 - e =: \text{work_triangle_length} \tag{5.5}$$

We can enforce this restriction by padding the input time series with additional values and by taking care that the padded values do not modify the overall result. The padding mechanism will be explained afterwards, we assume for the moment, that the condition holds and partitioning is possible.

With the stated preconditions, we can express the partitioning by specifying the tile length and the offsets into the distance matrix of the upper leftmost distance entries of the lower

tiles as follows:

$$\text{tile_row} = \lceil \sqrt{\text{tile_id}} \rceil \tag{5.6}$$

$$\text{tile_col} = \lfloor (\text{tile_id} - (\text{tile_row})^2)/2 \rfloor \tag{5.7}$$

$$\text{tile_length} = \text{work_triangle_length}/\sqrt{p} \tag{5.8}$$

$$\text{start_row_offset} = e + \text{tile_row} * \text{tile_length} \tag{5.9}$$

$$\text{start_col_offset} = \text{tile_row} * \text{tile_length} \tag{5.10}$$

While the tiles have sides of equal length, the length of the exclusion zone breaks symmetry in the computation of the offsets. Figure 15a illustrates the reason: as the working triangles left border is aligned with the distance matrices left one, the exclusion zone need not be added to the starting offset. In vertical direction though their bottoms are aligned to each other. This alignment is also reflected in the offsets of the required input sub-series as well the offsets of the partial result slices produced by the tile. Figure 15a marks the required input sections for tile 7 as an example. The starting offsets of the input subsequences are exactly aligned to the row and column offset of the tile. The same holds for the respective output sections. While tile 7 resides in the second column and third row, the offset due to the exclusion zone produces a small gap in the profile output sections generated by row- and column-wise projections.

Input padding

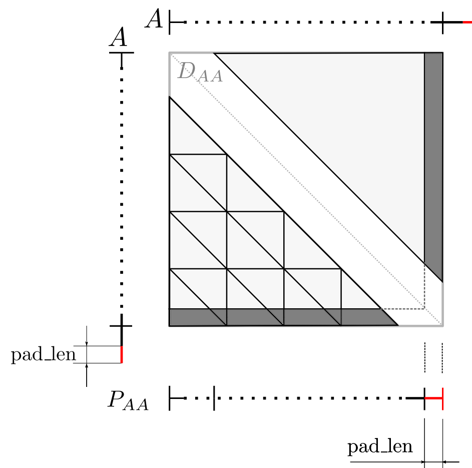


Figure 16 Padding of the time series input: The described tile partitioning scheme can be applied only, if condition 5.5 is met. This can be achieved by appending pad_len values to the time series. This padding extends the distance matrix by virtual entries, shaded in dark gray, which will be evaluated in the computation and generate a prolonged matrix profile. Careful modification of meta-series like the sliding mean ensures, that the output is not modified compared to a profile without the padding, after truncation of additional pad_len values.

If the length of the input time series does not meet the condition in equation 5.5, application of the tiling scheme is not possible immediately. In order to apply it, we extend the input series by appending the minimum required amount of $\text{pad_len} \in \mathbb{N}$ zeroes to it, such that the length

condition holds.

As illustrated in 16, the padding of the input series causes a growth of the distance matrix and its matrix profile. As the purpose is to obtain the matrix profile of the input result, it must be ensured, that the additional distance entries, shaded in dark gray, do not modify any values of the matrix profile and the result needs to be truncated. Result truncation is achieved, when writing to the file by dropping the values. Enforcing that the artificially created entries of the distance matrix take on maximum values ensures that they do not modify the resulting profile, as the matrix profile searches for minimum distances entries in the matrix according to equation 4.3.

Maximum distance values, or respectively minimal score values in terms of the optimized kernel according to equation 4.5, are computed in an unmodified evaluation kernel, if the precomputed $\widetilde{\mu}_i$ and s_i coefficients for the padded time series entries are equal to 0, as shown by equation 4.9. For this reason we invalidate during the precomputation the last $\mathrm{pad_len}$ values of the $\widetilde{\mu}$ and s meta-time-series by setting them to 0, instead of computing the regular values from the sliding mean and standard deviation as defined in equation 4.9.

With the modified $\widetilde{\mu}$ and s coefficient, the padding does not have any further impacts on the algorithm until finally writing to the file, where the result needs to be truncated. Neither the kernel, nor the iteration scheme nor the communication needs to be adapted if the matrix profile is computed over such a prolonged time series.

As the padding adds artificial distance entries to the matrix, additional time is spent for evaluation of these entries, causing a runtime overhead. As the tiling scheme for a square number of p processes consists of $\sqrt{p}$ columns and we pad by the least possible amount, the amount of padded time series samples is bound by: $\mathrm{pad_len} < \sqrt{p} - 1$. Given a input problem of a time series and subsequence window length, which produces a matrix profile of length $\mathrm{profile_length} \in \mathbb{N}$, an estimate for the upper bound of the relative runtime overhead compared to a computation without the padding can be stated based on the fraction of added matrix entries:

$$\begin{aligned} \mathrm{rel_padding_overhead} &= \frac{(\mathrm{profile_length} + \sqrt{p})^2 - \mathrm{profile_length}^2}{\mathrm{profile_length}^2} \\ &\approx \frac{2\sqrt{p}}{\mathrm{profile_length}} \end{aligned} \tag{5.11}$$

As the $\mathrm{profile_length}$ exceeds the number of processes by several orders of magnitude in relevant scenarios, e.g $p \approx 1 \cdot 10^3$ and $\mathrm{profile_length} \approx 1 \cdot 10^8$, this upper bound is in a range below $0.1\,\%$ and dominated by other overheads, as we will see in the experimental results.

Tile evaluation kernel

For evaluation of the tiles, the discussed general kernel of section 4.2.1 with the blocked SCRIMP iteration scheme is used for the lower tiles. For evaluation of the upper triangular tiles the same kernel can be used if the horizontal and vertical in- and outputs are swapped, due to its symmetry. Figure 17 illustrates the idea: The horizontal result of a upper-triangular AB-kernel-invocation is equivalent to the vertical result of a lower-triangular BA-kernel invocation. Conversely the vertical result of the upper tile is obtained as the horizontal result of the lower triangular kernel invocation with the swapped inputs. The reason is, that swapping the inputs of a AB-join is equivalent to mirroring a distance matrix along its main diagonal. I.e. rows and columns are swapped and the horizontal result of looking for minima within columns in the AB-join is equivalent to a row-wise minimum search in the swapped matrix of the BA-join, i.e. its vertical result.

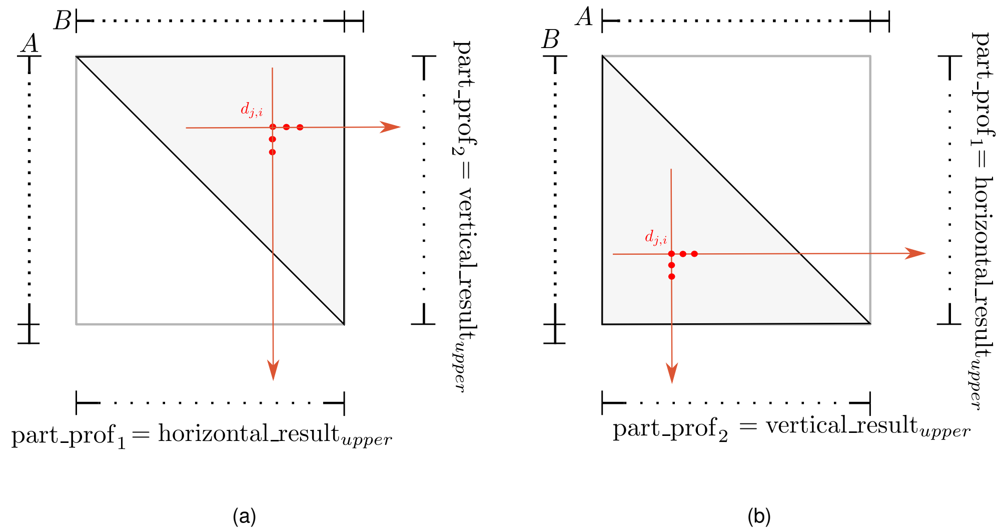


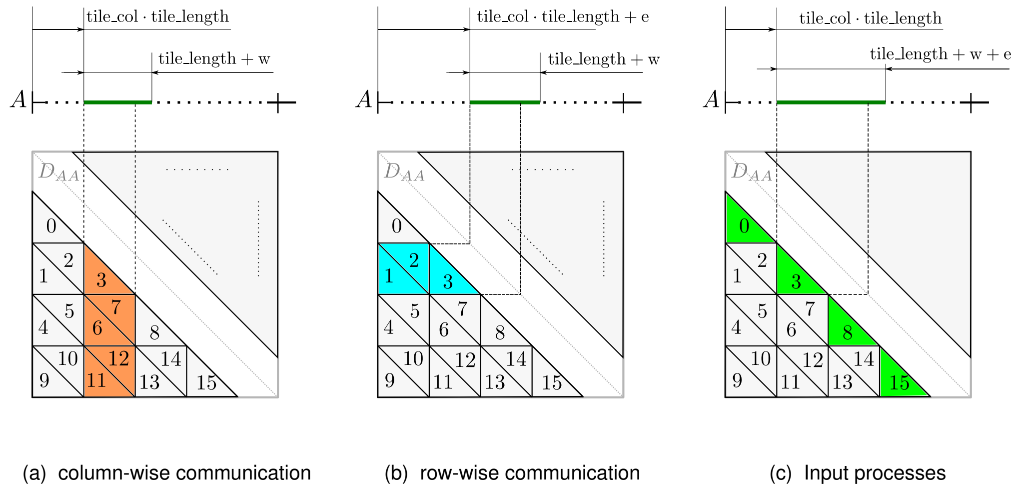
Figure 17 Evaluation of upper tiles: evaluation of a upper triangular tile as depicted in (a) is performed with the SCRIMP kernel of a lower triangular tile after swapping the horizontal and vertical input and output series

File input

Several of the tiles require to load similar sections of the input time series. Redundancy in the required input data is found along rows and columns in the tiling: For any $i \in 0, \ldots, p-1$, input processes in the i-th column of the input tiling require the same input subsequence. Mostly overlapping with that is the input subsequence of all processes in the i-th row. Figure 18 illustrates the input subsequences for the second row and column. Apparently, the two slices are offset relative to each other by the size e of the exclusion zone. As e is a fraction of the window length w, it is quite small compared to the tile length.

We selected the processes along the distance matrices main diagonal as input processes, depicted green in 18c. As their column and row indices are identical, both of their required

input slices are mostly overlapping and can be loaded by a single contiguous read of length $\mathrm{tile_length} + \mathrm{e} + \mathrm{w}$ at offset $\mathrm{tile_col} \cdot \mathrm{tile_length}$. After reading, the input processes perform two sending operations: The first $\mathrm{tile_length} + \mathrm{w}$ of the read samples are sent to all the processes within the same column. The same amount, starting at an offset of e in the file input buffer, is sent to all processes within the same row.



(a) column-wise communication (b) row-wise communication (c) Input processes

Figure 18 Input partitioning and communciation: all tiles withing one column require the same input data, as shown for the second column in (a). The same holds for rows, as depicted in (b). The required input sections for rows and columns differ by the length of the exclusion zone e. Shaded green in (c) are all the tiles along the distance matrices main diagonal, whose processes are chosen to perform the file reading. By reading a sufficiently large input subsequence, as illustrated for process 3, the respective subsequences can be spread along rows and columns.

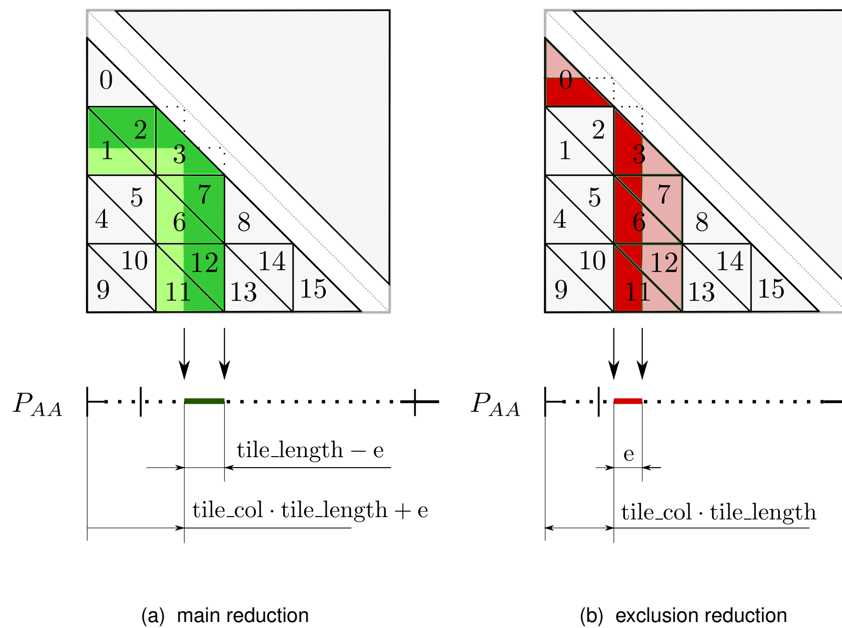
It shall be noted, that the proposed solution does not completely avoid redundant reads. To be exact, subsequent input slices in the file overlap by the sum of the window and exclusion zone length $\mathrm{w} + \mathrm{e}$. As those are only a small fraction of the total input size and the disk accesses are sequential, we argue that the resulting overhead is small enough to be neglected.

Result reductions

After the processes have finished the independent evaluation of their respective tiles, each one holds two intermediate result slices. Slices which contribute to the same section of the global result need to be merged and the final result communicated to the specific process which is responsible for writing the output. To perform the file writing, we chose the same processes as for reading: namely the lower tiles closest to the diagonal of the distance matrix, as shown in figure 18c.

The merging process can be defined as a reduction operation by defining the merging operation of two slices. The operation is mostly the same as in the trivial parallelization, as defined in algorithm 3. Instead of specifying the full profile length, the length of the slices to be merged is passed in together with their buffers.

Before stating the mathematical definition of the slices and the participating processes, let us review the design, visually outlined in figure 19. As previously pointed out, the exclusion zone in the distance matrix causes asymmetry in the alignment of the output slices. We identified unique groups of processes contributing to mutually exclusive sections of the global result. For that purpose, we logically split each processors result slices again into two parts. Figures 19a and 19b illustrate two slices of the global result P_{AA}, which are aligned with the horizontal tile partitioning of the distance matrix. Partial results contributing to the sections are produced by the colored processes, marking sections of the distance matrix with a darker coloring. Target process of both depicted result reductions is process 3. Note that the size of the exclusion zone is exaggerated in the figure. As e is very small compared to the tile length, the first result slice (fig. 19a) of length $\mathrm{tile_length} - \mathrm{e}$ summarizes most of the results within the tiles. For that reason we call the corresponding communication operation the *main reduction*. The second slice (fig. 19b) covers a comparatively small fraction caused by the misalignment due to the exclusion zone, after which we name the according reduction as the *exclusion reduction*.

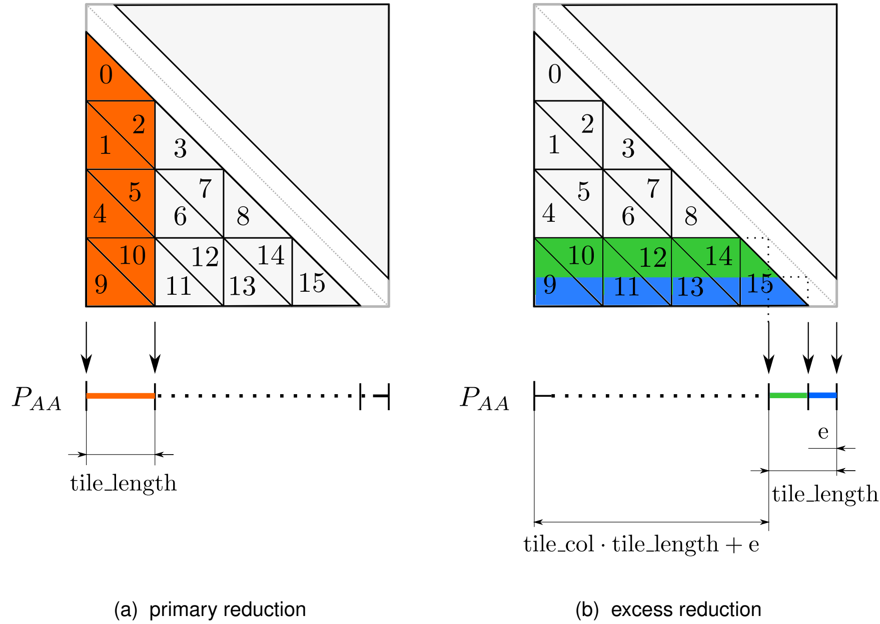


(a) main reduction (b) exclusion reduction

Figure 19 Merging partial result slices: Illustrated are two exemplary communication groups of processes which produce mutually exclusive slices of the global matrix profile P_{AA} coarsely aligned with the second column of the tiling scheme. The slices are produced by merging of local partial matrix profiles. Two different communication groups are formed for each column in the tiling scheme. Highlighted with colors are the two communicators for the slices of the second column. Colored in dark are the respective sections of the distance matrix, which contribute to individual slices.

Construction of all $2 \cdot \sqrt{p}$ such result slices, one of each kind for every column of the partitioning scheme, does not yet cover the full global matrix profile length. The virtually evaluated upper triangle in the distance matrix produces another slice of e profile values after the projection of the last tiles rightmost matrix entry onto the profile, as depicted blue in figure 20b. Computation of the respective values is performed by all the tiles in the last row of the tiling scheme. A reduction of the last e values of their vertical result slices, yields the missing slice.

The process group of the main result reduction for the very last result column degenerates to the same single row group, as depicted in green in figure 20b. We combine both operations by performing a single reduction of $\mathrm{tile_length}$ matrix profile values, which we call the excess reduction. Similarly, the process groups of the main and exclusion reduction of the first column degenerate to a single column and we combine them, as depicted in illustration 20a again to a reduction of a single slice of length $\mathrm{tile_length}$. We treat this case and the previously explained excess reduction as special cases of the main reduction in our formal definitions.



(a) primary reduction (b) excess reduction

Figure 20 Special cases of result merging: Subfigure (a) depicts the combination of main and exclusion reduction (see fig. 19), which is possible in the very first column, as the horizontal parts of the process groups are missing. Subfigure (b) shows in blue the additional reduction required for the very last slice of the global result, as caused by the asymmetry due to the exclusion zone. As the communicator structure is identical to that of the main reduction (see 19a) for the last column, we handle the excess by extending the length of the main reduction, which is shown in green.

Let us start with the formal definition of the exclusion reduction, which is exemplified in fig. 19b for the slice aligned with the second column in the tiling scheme: We parameterize the definition by the column of the tiling scheme, with which the resulting slice is aligned, denoted as $\mathrm{result_col} \in \{1, \ldots, \sqrt{p} - 1\}$. The starting offset and length of the slice within the global matrix profile result P_{AA} are given as:

$$\begin{aligned} \mathrm{slice_offset} &= \mathrm{result_col} \cdot \mathrm{tile_length} \\ \mathrm{slice_length} &= \mathrm{e} \end{aligned}$$

Depending on its location in the tiling scheme, a process with tile-ID $\mathrm{tile_id} \in \{0, \ldots, p - 1\}$

contributes following slices of its local results to the exclusion reduction of $\mathrm{result_col}$:

$$\mathrm{vertical_result}[\mathrm{tile_length} - \mathrm{e}, \ldots, \mathrm{tile_length} - 1] \text{ iff}$$
$$\mathrm{tile_row} = \mathrm{result_col} - 1 \wedge \mathrm{result_col} > 0$$
$$\mathrm{horizontal_result}[0, \ldots, \mathrm{e} - 1] \text{ iff } \mathrm{tile_col} = \mathrm{result_col} \wedge \mathrm{tile_col} > 0$$

Note, that no exclusion reduction is defined for $\mathrm{result_col} = 0$: The respective slices are handled as a extended main reduction (fig. 20a). A further observation is that all tiles, except for those in the last row, participate in two reductions of different result columns. E.g. tile 6 participates in the reduction of $\mathrm{result_col} = 1$ as part of the vertical leg (as depicted in illustration 19b), but also in the reduction of $\mathrm{result_col} = 3$, as part of the horizontal leg.

We define the main reduction, exemplified for $\mathrm{result_col} = 1$ in 19a, again depending on the column in the tiling scheme under which the resulting profile is aligned. Starting offsets and length of a slice within the global matrix profile result P_{AA} are computed as:

$$\mathrm{slice_offset} = \begin{cases} 0 \text{ iff } \mathrm{result_col} = 0 \\ (\sqrt{p} - 1) \cdot \mathrm{tile_length} \text{ iff } \mathrm{result_col} = \sqrt{p} - 1 \\ \mathrm{result_col} \cdot \mathrm{tile_length} + e \text{ else} \end{cases}$$
$$\mathrm{slice_length} = \begin{cases} \mathrm{tile_length} \text{ iff } \mathrm{result_col} \in \{0, \sqrt{p} - 1\} \\ \mathrm{tile_length} - \mathrm{e} \text{ else} \end{cases}$$

A process with tile-ID $\mathrm{tile_id} \in \{0, \ldots, p - 1\}$ and the location $(\mathrm{tile_row}, \mathrm{tile_col})$ in the tiling scheme, contributes following slices of its local results to the main reduction of column $\mathrm{result_col}$:

$$\mathrm{vertical_result}[0, \ldots, \mathrm{tile_length} - \mathrm{e} + 1] \text{ iff } \mathrm{tile_row} = \mathrm{result_col} \wedge \mathrm{tile_row} > 0$$
$$\mathrm{horizontal_result}[\mathrm{e}, \ldots \mathrm{tile_length} - 1] \text{ iff } \mathrm{tile_col} = \mathrm{result_col} \wedge \mathrm{tile_col} \in \{1, \ldots, \sqrt{p} - 2\}$$
$$\mathrm{horizontal_result}[0, \ldots \mathrm{tile_length} - 1] \text{ iff } \mathrm{tile_col} = \mathrm{result_col} \wedge \mathrm{tile_col} \in \{0, \sqrt{p} - 1\}$$

Notably the processes along the main diagonal contribute parts of both, their vertical and horizontal result, to the same reduction. All other tiles participate in reductions of two distinct slices. E.g. tile 6 contributes to the main reduction of $\mathrm{result_col} = 2$, as highlighted in the illustration, but also to the main reduction with $\mathrm{result_col} = 3$.

File output

All the result slice reductions are implemented such that the chosen output processes along the main diagonal receive the merged result slices. Every output process receives the result of a main slice reduction and a exclusion reduction. Figure 21 illustrates the output processes and the respective output slices for process 3 and 15. For a given output process with column

$\text{tile_col} \in \{0, \dots p-1\}$ in the tiling scheme, the respective section of the matrix profile to be written to the file is:

$$\text{slice_offset} = \text{result_col} \cdot \text{tile_length}$$

$$\text{slice_length} = \begin{cases} \text{tile_length iff result_col} < \sqrt{p} - 1 \\ \text{tile_length} + \text{e iff result_col} = \sqrt{p} - 1 \end{cases}$$

As the slices do not overlap and are subsequent in the output series, they are received in a single buffer which is written to the file in parallel to all the other processes.

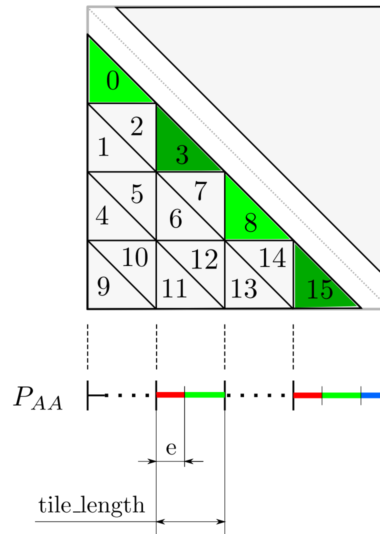


Figure 21 Output processes and partitioning: The lower triangular tiles along the main diagonal of the distance matrix are responsible for file output and colored green. The respective sections of processes 3 and 15 are highlighted. They are colored according to the result reductions of figure 19 from which they originate.

5.2.2 Mapping to the MPI

File input

As outlined in the previous section, a set of input processes is given by the processes assigned to the tiles along the main diagonal of the distance matrix, as illustrated in figure 18c. We construct a MPI communicator with theses processes with the help of the group mechanisms [67, pp. 228 sqq.]. With the communicator, we read in the described input subsequences in a collective blocking MPI_File_read_all call from the same binary file as described in section 5.1.2. Again we use a file view, created as a linear sub-array of the data section with the MPI_Type_create_subarray call. The required information about the full length of the time-series is obtained from a broadcast in MPI_COMM_WORLD by process 0, who reads it in a preceding blocking individual read of the header using MPI_File_read_at.

Input communication

Spreading of the respective input subsequences to the processes with tiles in the same column and row, as illustrated in figure 18c, is realized as a broadcast to appropriate communicators. Every process participates in construction of one row and one column communicator through a MPI_Group_incl call. To reduce synchronization overheads, both broadcasts are realized as a asynchronous MPI_Ibcast.

Merging partial results

Merging of partial result slices, as illustrated in 19, is achieved by collective reduction operations. A custom reduction operator is specified for pairwise merging of profiles, as already described for the trivial parallelization in section 5.1.2. For each of the reductions, communicators are set up with the group mechanisms [67, pp. 228 sqq.]. I.e. the communicators for the main- and exclusion reduction with their angular structure (fig. 19a, 19b) are built as the union of appropriate row and column groups. The rank of the receiving output processes along the main diagonal within the new communicators is determined from their known world rank with MPI_Group_translate_ranks.

As already pointed out in the description of the communication structure, most tiles participate in two main and exclusion reductions, once as part of the horizontal and once as part of the vertical leg of the angular communicator structures. We use the asynchronous reduction collective MPI_Ireduce, in order to avoid deadlocks, as no lock-free ordering exists. Special handling is further required for the output tiles along the main diagonal: their processes participate only in a single main slice reduction but contribute both of their local results to it. As only a single collective reduction call with a single data argument is permissible, the two local results are merged locally in advance to the reduction operation by a manual invocation of the merging operation. Calls to MPI_Wait ensure completion of the reductions before results are written to a file.

Result file output

As the sets of output and input processes are identical, we reuse the input communicator for parallel writing of the output file. We employ the same output file format as for the trivial parallelization, described in section 5.1.2 and perform writing of the slices again with the blocking collective MPI_File_Write_all with the previously explained partitioning. The binary file header is written by the process with rank 0 within the communicator with a individual file write.

5.2.3 Theoretical Performance Analysis

A full performance analysis of all program sections and implementation details is untenable. For simplification we focus only on the major program sections an apply some simplifica-

tions.

First of all we assume the communication and I/O behaviors to be dominated by the bandwidths and ignore any latencies. This is reasonable to assume for sufficiently large input sizes, as all messages grow with the input lengths. We further ignore impacts of the window length and the exclusion zone, which is valid under the typical assumption of small window lengths compared to the overall problem size [1]. In particular we assume the input an matrix profile to be of approximately the same length n. Furthermore, the exclusion reduction is completely ignored: the message lengths with the size of the exclusion zone are fixed and is performed asynchronously to the main result reductions. As the main reductions exhibit a very similar communication structure but operate with significantly larger messages, their runtime behavior is dominating.

Table 5 lists the total runtime behavior of program sections we deemed most interesting. The middle column lists the behavior as derived for the input length n and p processors. Due to our simplifying assumptions, the tile length of each partition equals $n/\sqrt{p}$.

As each processor needs to precompute meta series for time series sections of the tile length, approximately $p \cdot n/\sqrt{p}$ time is spent for the those. Broadcasting all input sections is performed with messages of the same length $n/\sqrt{p}$. The largest input communicator is composed of a full row/column of $2\sqrt{p} - 1$ tiles, such that broadcast takes approximately $\log(\sqrt{p})n/\sqrt{p}$ time[44, p. 188]. We can assume all p processes to spend that minimum amount of time for the broadcast, also for smaller input communicators, due to the implicit synchronization.

Table 5 Theoretical scaling analysis of distributed implementation: simplified scaling behavior of important program sections. The middle column states the approximate runtime behavior depending on the number of processes p and the input length n. The rightmost column list simplified expressions depending on the problem size $W \propto n^2$. *cost* refers to the accumulate time over all processes spent in distinct code sections

functionality	**cost** $T(p, n)$	**cost** $T(p, W)$
precomputations	$\Theta(p \cdot n/\sqrt{p})$	$\Theta(\sqrt{p \cdot W})$
kernel evaluations	$\Theta(n^2/2)$	$\Theta(W)$
input broadcast	$\Omega(p\log(\sqrt{p}) \cdot n/\sqrt{p})$	$\Omega(\sqrt{W \cdot p}\log(p))$
main reduction	$\Theta(p\log(\sqrt{p}) \cdot n/\sqrt{p})$	$\Theta(\sqrt{W \cdot p}\log(p))$
File I/O	$\Omega(p \cdot n/\sqrt{p})$	$\Omega(\sqrt{p \cdot W})$

The main reductions are performed in communicators of $\sqrt{p}$ processes. Under our simplifying assumptions each reduction produces a result slice with a length equal to the tile-length

$n/\sqrt{p}$. As all p processes participate in such reductions (each in two), we end up with the expression in the table.

File input and output operations of the I/O processes apply to an equal amount of $n/\sqrt{p}$ elements, if we ignore w and e. For this reason the written and read data amount considered proportional for our analysis and merged as they show the same behavior. While there are only $\sqrt{p}$ processes performing actual I/O operations, the algorithm implies that all other workers are idle. We subsume this idle time under general file I/O. Together with the data length (assuming a bandwidth dominated behavior), this yields a $p \cdot n/\sqrt{p}$ behavior. It is a lower bound, because the underlying MPI implementation potentially contains inter-process communication and furthermore network structure and file system [68] can impact the scaling behavior.

As we ignore the exclusion zone for our assumption, the distance matrix has $\approx n^2$ entries. Because the implementation does not perform redundant evaluations, this is the number of total kernel evaluations. As the kernel has a fixed amount of FLOPs, it is proportional to the arithmetic work of the algorithm and constitutes the problem size W of our algorithm for a iso-efficiency analysis [62]. The rightmost column in the table shows the simplified complexities rewritten in terms of W.

Summing up all the terms in the rightmost column (tab 5) and adding constants, we obtain the following theoretical best-case runtime model for our implementation:

$$T_{\text{total,theo}}(p, W) = c_1 + c_2 \cdot W + c_3 \cdot \log_2(p)\sqrt{p \cdot W} + c_4 \cdot \sqrt{p \cdot W} \tag{5.12}$$

In section 6.3.1 we try to apply it to the empirical behavior in experiments. At this point we can observe the theoretically dominating terms in different scenarios: when increasing the input lengths with a fixed number of processes the fastest growing time is that of the kernel evaluations $c_2 \cdot W$. This promises, that computations become more efficient with growing problem sizes, as the impact of overheads vanishes. In a strong scaling scenario, the long term dominating overhead based on this theoretical model is the communication.

Based on the listed behaviors, we can state the isoefficiency function [62] of our implementation. The dominating behavior is obtained when balancing the problem size against the input broadcast or result reductions as follows:

$$\begin{aligned} W &= K\sqrt{W \cdot p}\log p \\ \implies n^2 \propto W &= K^2 p \log^2 p \end{aligned}$$

The isoefficiency metric for the algorithm is $\Theta(p\log^2(p))$. It is beyond the class of ideally scalable systems [62] with $\Theta(p)$. To maintain a constant efficiency for cost optimal scaling, the input length n needs to grow at a rate proportional to $\sqrt{p}\log p$

5.2.4 Comparison to SCAMP

Concurrent to our work Zimmerman *et al.* [2] present a cluster parallelization of the matrix profile algorithm, which allows computations of matrix profiles beyond the memory limit of single processing units and scaling to arbitrary numbers of workers. In contrast to our work, their target cluster is the Amazon AWS cloud service and specifically GPUs are employed as accelerators.

Computation is realized by batch processing, as illustrated in figure 22, utilizing a very similar work-partitioning. Input segments for the different tiles are stored on the cluster. The different tiles form a queue and are assigned to available workers in a round-robin fashion. Intermediate results of individual workers are stored on the cluster again, until processing of all tiles is finished. Afterwards a single instance is used to merge all the local results to produce the final matrix profile.

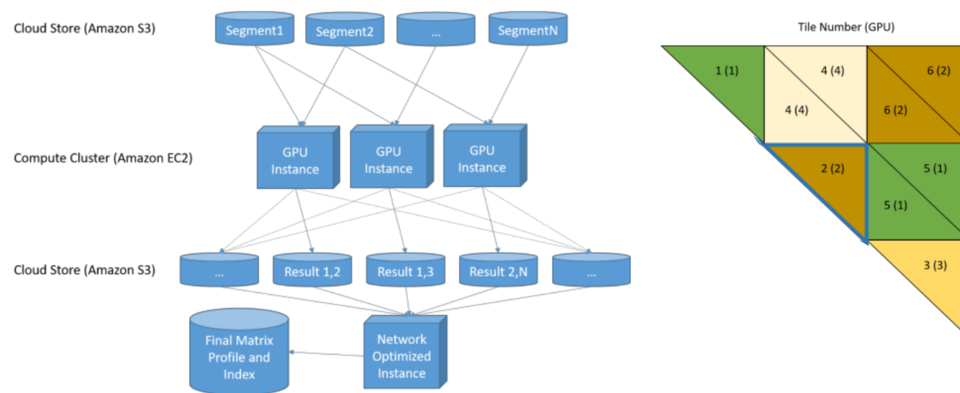


Figure 22 SCAMP processing scheme, images by Zimmerman *et al.* [2]: SCAMP considers evaluation of the upper triangular part of the distance matrix for a self-join, which due to symmetry makes no difference. Shown on the right hand side is the partitioning and distribution of distance evaluations to different workers, i.e. GPUs. Triangular tiles form a queue according to the tile number and are subsequently scheduled to workers (4 GPUs in the depicted example). The left part of the illustration shows the batch processing pipeline on a Amazon AWS cluster.

The outlined processing scheme, in contrast to our work, avoids any explicit inter-process communication. As the intermediate results are stored on disk and merged later on, the file system is used as a communication substitute. Also different to our approach reading of input data is always performed redundantly, such that more disk accesses are required. As pointed out by Zimmerman *et al.* [2], the granularity or number of tiles determines the amount of required storage space for intermediate results. Namely usage of r rows of tiles requires r times the space of the final matrix profile for the intermediate results. For example they report a intermediate data size of 196.4 GB for computation on a $1 \times 10^9\,\mathrm{sample}$ input series with 40 rows of tiles.

SCAMPs tiling scheme, as depicted in figure 22, is like ours based on a triangular decomposition. In contrast to ours a square of two neighboring triangles is assigned to a single worker

instance, wherever possible. This is done to avoid some of the redundancy in loading and storing, as such tiles operate on the same input and result sections. The tile size is chosen such that a worker achieves the highest throughput (see sec. 6.4.2). The number of workers in contrast to our work is arbitrary, as tiles form a linear queue which is assigned to available workers in a round-robin fashion.

Obvious advantages of the approach compared to ours are, that the batch-queue does not require a distinct number of available compute resources but can adapt to the available cluster resources. Furthermore it provides fault-tolerance, as computation of single failed instances can be restarted.

6. Experiments

The following subsections describe experiments and results conducted to examine the performance of the implementations outlined in sections 4 and 5. The chosen metrics and techniques which we apply are briefly review in section 3.3. We adopted the SCRIMP C++ code of Zhu *et al.* [63] in our framework as a sequential baseline implementation and used it to validate correctness of our implementation by comparing matrix profiles computed on the same input data with the different implementations and different processor configurations.

If not stated differently, presented measurements are obtained on the Supermuc Phase 2 cluster [69]. Relevant hardware details, versions of used libraries and build tools used on the system are listed in appendix 8.1. Reported there are also results of the Hpcbench [70] benchmark for MPI point-to-point communication to assess the network performance of the system.

Following the outline of our theoretical explanations, we start in section 6.1 with experiments to investigate the contributions of our sequential kernel optimizations, comparing to the baseline algorithm of Zhu *et al.* [32]. Afterwards scalability of the trivial parallelization is examined in section 6.2. The first experiment reported for the distributed parallelization (sec. 6.3) is fitting of two runtime models to empirical data. A detailed investigation of scalability and respective bottlenecks provides insights into the fitting behavior. In section 6.4 a strong scaling experiment is used to compare implementations to each other as well as to demonstrate low instrumentation overhead to justify our in-depth investigations. Finally an experiment reported by Zimmerman *et al.* [2] is repeated with our implementations to compare our contributions to the SCAMP framework (sec. 6.4.2).

6.1 Sequential Optimization

6.1.1 Comparison of Kernels

Figure 24 compares different kernels based on their computational throughput in sequential program runs. The measured time considers only computational work for processing the distance matrix and obtaining the matrix profile. I.e. the start time measurement is taken after all buffers are initialed and the precomputations of meta-time-series (e.g. the sliding mean of the original SCRIMP implementation) are finished. Stop time is acquired, after the final matrix profile is held in memory, i.e. the post-processing required due to the arithmetic kernel optimizations is contained in the time. Throughput is computed by dividing the number of evaluated distance entries (including the mirrored ones of the upper triangle in the distance matrix) by the respective time difference. Subsequence search was performed with a window length of 10×10^3 with two random walk time series of lengths 100×10^3 and 1×10^6 samples. Both contained several randomly embedded motifs with a length equal to the chosen

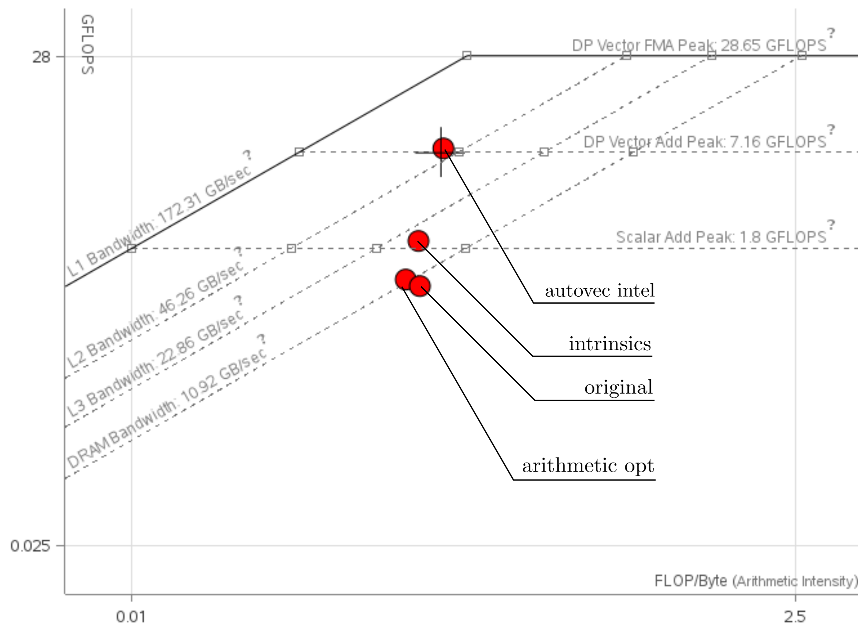


Figure 23 Roofline diagram comparing different kernels: The red dot indicates performance of the innermost loops iterating over entries in the distance matrix. Note that both axis are scaled logaritmically. Numerical values of the shown data points as reported by Intel Advisor 2018 are listed in table 6.

search window size. According to the estimate of section 4.2.2 variables accessed during computation of the smaller problem take up 4.8 MB of memory and respectively 48 MB in case of the larger input size. As we perform each sequential run on a exclusive compute node of the SuperMUC Phase II target system, full 18 MB memory of level 3 cache are exclusively available to a single processor. I.e. the size of the large problem is chosen such that the required data do not fit into the cache, in contrast to the small one.

Table 6 Roofline data points of kernels: data reported by Intel Advisor 2018 for the roofline diagram in figure 23. The single experiment for data recording is a self-similarity search of a input series of length 1×10^6 with a search window length of 1×10^3 samples. The bandwidth considers the total amount of transferred data between CPU and memory subsystem, i.e. also includes traffic to the caches

kernel	**arithm. intensity** / FLOP/B	**performance** / GFLOPS	**bandwidth** / $\mathrm{GB\,s^{-1}}$
original	0.112	1.043	9.283
arithm. opt	0.100	1.146	11.463
intrinsics	0.111	1.995	17.950
autovec	0.136	7.553	55.347

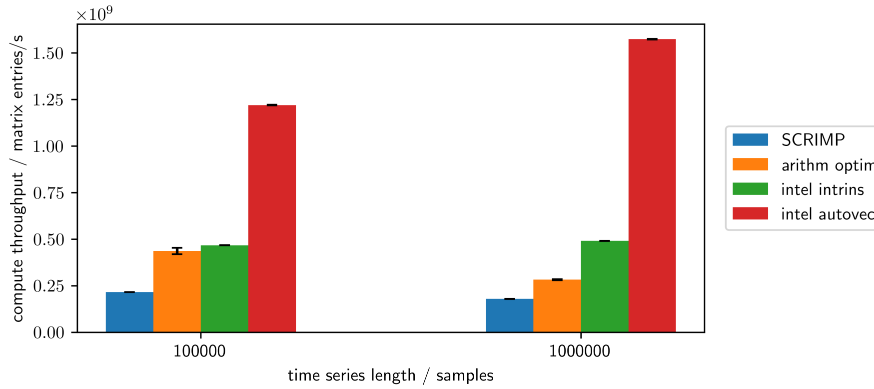


Figure 24 Comparison of sequential kernels: Shown is the throughput of different sequential kernel version for two problem sizes, each time performing search with a window length of 10×10^3 samples. Comparison is based on the computational throughput: the number of distance matrix entries was divided by the time spent purely for computations. The first kernel shows the original SCRIMP kernel from Zhu *et al.* [32]. The second shown kernel (*arithm optim*) applies the arithmetic optimizations of section 4.2.1 without modification of the iteration scheme. *intel autovec* depicts the performance of the vertically blocked self-join-kernel, which the Intel compiler was able to vectorize automatically and *intel intrins* shows the performance of the manually vectorized general AB-join implementation. For each data point 5 measurements had been performed and the most important statistics are appended in table 18. Note that the standard deviations are hardly noticeable, as they are at least one order of magnitude smaller than the actual values

Figure 24 shows the throughput of four different kernel versions. The original SCRIMP kernel of Zhu *et al.* [32] serves as a baseline using the original publications source code [63] with minor adaptations to integrate it in our custom framework. The second kernel variant shown, *arithm opt*, demonstrates the impact of the arithmetic optimizations presented in section 4.1 without modification of the iteration scheme. For the blocking iteration scheme, two versions with different vectorization are shown: *intel autovec*, the self-join kernel auto-vectorized by the Intel compiler, and *intel intrins*, the manually vectorized intrinsics kernel. For both blocked kernels, the measurement was performed with a block length of $500\,\mathsf{entries}$. This choice was made according to a experiment with varying block-lengths, which is presented in subsection 6.1.2. To support our argumentation, figure 23 additionally shows a roofline diagram generated with Intel Advisor 2018, which contains respective roofline points for each kernel. The roofline diagrams were obtained in a single experimental run with the $1 \times 10^6\,\mathsf{sample}$ series problem, which requires DRAM accesses.

The throughputs in figure 24 show that the arithmetic optimizations accelerate computation by a factor of 2 compared to the baseline for the small problem. For larger problem sizes the throughput (and accordingly the gained speedup) is decreased. A look at the roofline diagram reveals that the DRAM bandwidth is limiting the kernel. While both kernels operate at similar FLOP rates, the arithmetic optimizations still result in a increased throughput, as less floating point operations are required for evaluation of a single distance entry.

Limitations due to the DRAM accesses are overcome by the blocked iteration scheme, as the *autovec* and *intrinsics* kernel show: instead of a decrease when the required memory exceeds the cache size, even an increase in the throughput can be observed. This observation is an artifact of our experiment: in the measured computation time also the initialization of the

dot products is contained. Initialization of the diagonals is noticeable for the small problem, as the chosen window length of 10×10^3 constitutes $10\,\%$ of the problem size. As it becomes neglectable for the larger problem, we even observe a slight increase in throughput.

The roofline diagram shows that the auto-vectorized kernel operates close to the ridge point of the Level 2 cache and the double precision vector add peak. Obviously the blocking scheme successfully overcomes the bandwidth limitations of DRAM and L3 cache. Still the kernel does not yet achieve maximum performance according to its operational intensity. Intels auto-vectorization report and the assembly show, that indeed some FMA instructions had been generated. We obtained measurements by manual source code instrumentation with LIKWID [71] adding acceptable instrumentation overhead, as listed in 7. They suggest, that kernel is bound by the compute roof: the measured bandwidth between L1 and L2 cache is more than one order of magnitude below the sustainable maximum reported in the roofline chart. Less than $0.1\,\%$ of CPU cycles spent waiting for L2 loads indicate, that also L2 cache misses do not pose a bottleneck. It needs to be noted, that the used counters only consider memory loads. For this reason still write accesses could constitute memory bottleneck. Looking at the assembly one observes that large parts of the computation are translated to pure vector multiplications and additions instead of FMA instructions. This is enforced by the mathematical structure of the kernel. For this reason we argue that most likely the auto-vectorized kernel is compute bound.

Another possible source of the performance gap is the use of unaligned memory operations, as also hinted at by the compiler. Drepper [54] reports on an older CPU architecture than that of our system, that unaligned memory operations can possibly cause slowdowns up to $400\,\%$, which is a similar amount to the kernels performance gap to the peak performance in the roofline diagram. As our kernel iteration over a column block starts at a row offset according to the dynamic exclusion zone length, our method of aligned allocation of the buffer bases does not generally allow aligned data accesses (see alg. 1 and fig. 11). As the exclusion zone length is set by a user parameter previous to data loading, the implementation could be improved in that regard by dynamic adaptation of the alignment. Molka *et al.* [72] suggest a series of performance counters, which one could use to disprove that memory bandwidths or latencies are limiting the kernel. As the auto-vectorized kernel as a pure self-join is not applicable for the distributed parallelization, we omitted further investigations of its limitation.

Regarding our intrinsics kernel implementation, figure 24 reveals, that our manual intrinsics kernel does not achieve the highest possible performance: it performs the same arithmetic operations and applies the same iteration scheme as the auto-vectorized kernel. Accordingly, the memory access-pattern should be the same, too. Still it achieves less than $30\,\%$ of the auto-vectorized kernels throughput. The entry in the roofline chart indicates low efficiency of the vectorization: the compute rate hardly exceeds the scalar compute roof and is far from the peak vector performance. Location in the roofline diagram and the independence of the

displayed throughput from the input length proof, that the blocking iteration scheme resolved the DRAM bottleneck of the original kernel.

The performance counters recorded with LIKWIDs source code instrumentation in table 7 clearly suggest that the bottleneck is related to a memory access, as the CPU is stalled while waiting for outstanding memory loads during $30\,\%$ of the cycles. As Molka *et al.* [72] point out, the underlying performance counter *CYCLE_ACTIVITY:STALLS_LDM_PENDING* also captures stalls for other reasons, if a load is outstanding at the same time. It could possibly also show delays caused by subsequent result dependencies of multiplications. According to their analysis, checking the *RESOURCE_STALLS:SB* counter value should reveal the bottleneck. Based on the fact, that only a neglectable fraction of cycles is reported as stalled due to L1 data cache misses (by the *CYCLE_ACTIVITY_CYCLES_L1D_PENDING* counter), candidates are writing memory accesses and issues not directly related to memory, like the mentioned arithmetic dependencies of subsequent instructions. Due to our time schedule we did not check it and leave it to future work. Still we used this kernel with its sub-optimal performance for our parallel implementations, as the distributed parallelization requires a general AB-join kernel and the intrinsics kernel is our best implementation at hand for such a kernel.

Table 7 Performance counter measurements of vectorized kernels: data had been recorded in two separate experiments with a input series of length 1×10^6 and window length 1×10^3, in order to collect counter values of the *CACHES* and *CYCLE_ACTIVITY* performance counter groups. The source code was manually instrumented with LIKWIDs [71] Marker API to record counter values of the kernel evaluations only (surrounding the outermost evaluation loop with measurement start/stop calls). A selection of the measurements is shown and the mean computational throughput $\mathrm{tp_{inst}}$ of the two experiments was added to validate low instrumentation overhead. Based on the reference throughput $\mathrm{tp_{ref}}$ from the kernel comparison (table 18) the overhead is computed as $\mathrm{overhead} = 100\,\%(\mathrm{tp_{inst}} - \mathrm{tp_{inst}})/\mathrm{tp_{ref}}$

	intel autovec	intrinsics
L1 to/from L2 bandwidth / MB/s	635	95.7
L2 to/from L3 bandwidth / MB/s	127	45.5
Memory bandwidth / MB/s	1.84	2.55
CPI	417×10^{-3}	1.09
Cycles without execution / %	4.18	30.1
Cycles without execution due to L1D / %	150×10^{-3}	116×10^{-3}
Cycles without execution due to L2 / %	182×10^{-3}	379×10^{-3}
Cycles without execution due to memory / %	3.32	30.0
Throughput / entries/s	1.46×10^9	495×10^6
Instrumentation overhead / %	7.97	6.97

6.1.2 Blocking Kernel Performance

As presented in section 4.2.2, in theory upper and lower bounds for a efficient blocking size can be derived. In practice cache utilization and conflicts are highly dependent on the allocated addresses of the variables, which is why we empirically validate our blocking approach and determine the best blocking size by running the program with different block lengths. Figure 25 shows the results for both blocked kernel variants for a fixed input time series length of 100×10^3. As the window length impacts the memory access pattern, as depicted in figure 11 we performed multiple runs for each block length with a set of window lengths: $w \in \{50, 100, 1000, 4096\}$. Included is a measurement with a window length of 4096 samples, as a representative of combinations with a potentially increased number of cache conflict evictions [54]. Impacts of the window length variation are observed as part of the standard deviation of the experiment. Each experiment was repeated 2 times. Thus each bar represents 8 program executions. The process was pinned to a single core of a exclusive node. Thus the lowest level cache was not shared with other programs and fully available to the experiments process.

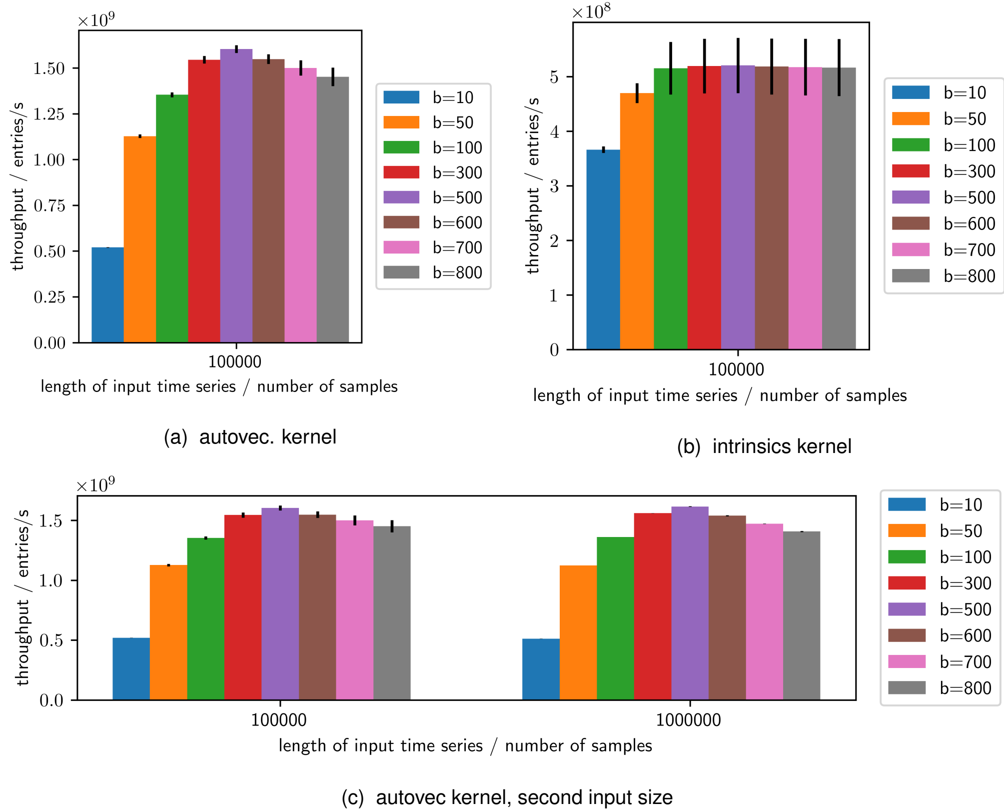


Figure 25 Variation of the block length b with the two blocked (and vectorized) kernels. Performance is measured as throughput of the iteration over the distance matrix: the number of evaluated matrix entries per second. The error bars indicate the standard deviation and capture variation of the input length. In different program executions the window length was varied, to check for impacts of the access pattern. Be aware of the difference in the charts vertical scaling. For figures (a) and (b) two experiments with four different window lengths are performed for each bar. Absolute runtime of the measurements with the small input series varied between 6.67 s and 27.99 s. To demonstrate independence of the behavior of the problem size, figure (c) shows additional measurements performed with 2 different window lengths for a larger input size. Runtimes for the large problem varied between 623.9 s and 1955.4 s. Numerical statistics of the measurement points are listed in tables 19 and 20.

The obtained results regarding the auto-vectorized kernel in figure 25a match the theoretical expectations: Peak performance is achieved for a block length of about 500 samples. Smaller block sizes have a lower throughput as cache locality is comparatively lower: similar to the arithmetically optimized kernel, to which the blocked one degenerates for $b = 1$, throughput is limited by the memory subsystem. Also with increasing block-length, the memory write of the vertical block result occurs more rarely, as outlined in section 4.2.2. The decreasing throughput for block-lengths above the peak originates from an increasing number of cache conflicts, as the overall cache size is limited. The results agree with the theoretically predicted cache exhaustion at a block length of 571 entries (see sec. 4.1). As those conflicts are highly dependent on the actual access pattern and allocated memory addresses, the standard deviation also increases with the number of cache conflicts. Up to block length with peak throughput, the standard deviation is neglegtably small. As the statistics are obtained from runs with varying window length, this proofs the independence of the kernels performance from the chosen window length.

The blocking behavior of the intrinsics kernel (fig. 25b) matches the observation of the roofline diagram, which was recorded with a block length of 500 matrix entries: in the limit, for a theoretical block length of 1, the iteration scheme is the same as as for the the arithmetic optimized one and the kernel is limited by DRAM accesses. With increasing block length, this limitation is overcome and the throughput increases. The increase stops soon at a block length of 300 entries, as the kernel ends up with the operating characteristics shown in the roofline diagram.

To proof the independence of the the behavior from the problem size, we further repeat the experiment with the auto-vectorized kernel and a input series length of 1×10^6 samples, which exceeds the available cache size. Because of the increased processing time, we used only two different window lengths in this experiment, namely $w \in \{1000, 4096\}$. As shown in figure 25c, the behavior is not affected by the problem size, which demonstrates effectiveness of the blocking approach.

According to this experiment we set the window length to $b = 500$ entries for all further experiments, as highest performance can be observed for both kernel versions. As the intrinsics kernel is the best one usable for the distributed parallelization, we will use it in all our following experiments, i.e. also for the trivial parallelization, in order to be able to compare the results. Notably the throughput of the auto-vectorized kernel suggests, that the employed intrinsics kernel could be further sped up by a factor of 3 by fixing the memory related bottleneck.

6.2 Trivial Parallelization

In order to provide a baseline and highlight the contributions of our distributed parallelization approach, we examine the scaling behavior of the trivial parallelization. We present the overall strong and weak scaling behavior for direct comparison later on (see section 6.4) and provide

more detailed investigation of the scaling bottlenecks to argue about the long-term validity.

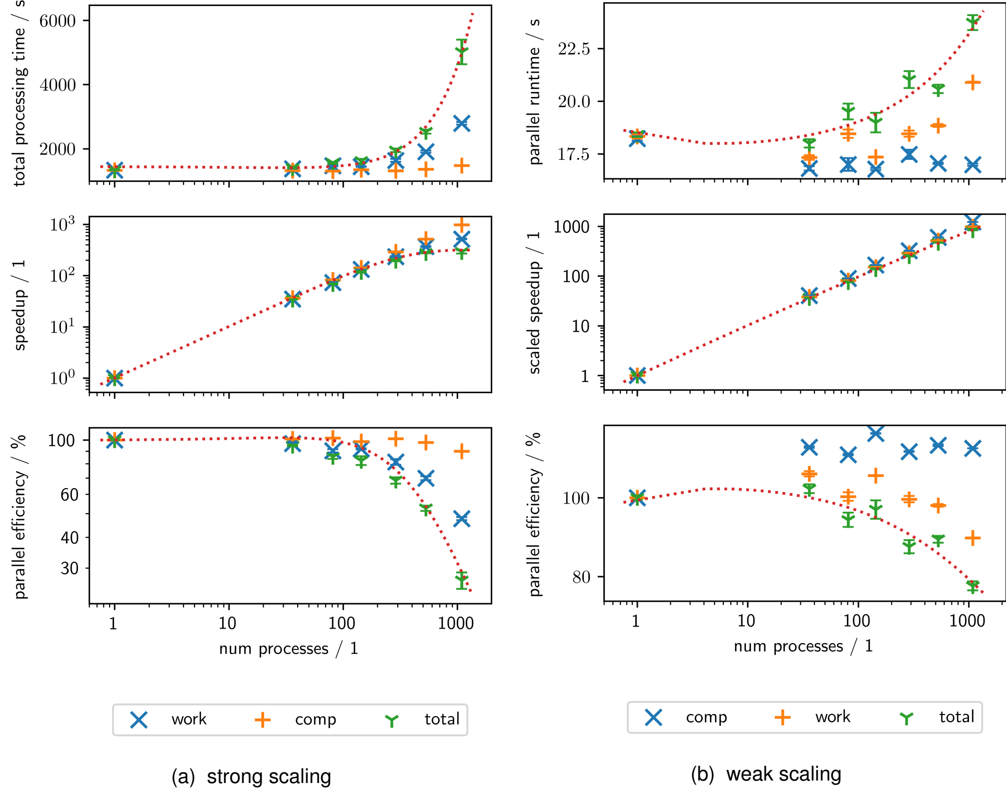
We use between 1 and 1089 processes in the experiments, which corresponds to 1 to 39 nodes. We apply an approximately logarithmic scaling scheme, using only square numbers of processors. While the implementation can be used with arbitrary numbers of processors, this choice was made to allow immediate comparisons to the results of the distributed implementation in section 6.3, which is restricted to square numbers. Further we omit scaling on a single node, as the inter-node communication shows a different behavior but is not of interest to us. We choose the input-lengths equal to the ones used in later experiments with the distributed parallelization (sec. 6.3) to allow comparisons. Runtimes of interesting program sections are tracked with manual time measurements with MPI_Wtime. Furthermore we use Score-P [60] instrumentation to collect runtime profiles of the application. To keep the overhead and profiling data size low, we apply a filter, which includes only functions in our relevant source files and disables tracking of all C++ standard library functions (appended in listing 8.1).

Figure 26b shows an overview of the trivial parallelizations' scaling behavior obtained from manual time measurements in the source code. The *total* measurements track the parallel runtime of the complete algorithm. To obtain consistent results, the start time is measured after a synchronization barrier. For this reason MPI initialization, command-line parsing, and varying startup times of different processes are excluded from the measurement. The stop time is measured after the matrix profile is written to the result file and measured times are logged afterwards. From the parallel runtimes $T_{\mathrm{par},i}$ logged by process i, the cost of the algorithm is obtained as $T_{\mathrm{total}} = p \cdot \max_i(T_{\mathrm{par},i})$. For analysis of specific code sections, we accumulate the time spent in them over all processes as $T_{\mathrm{accu}} = \sum_{i=1}^{p} T_{\mathrm{par},i}$. Average times in the weak-scaling scenario are obtained from the cost and accumulated times by division by the number of used processes: $T_{\mathrm{avg}} = T_{\mathrm{accu}}/p$

The *comp* time in figure 26b contains the time spent in precomputations of the meta-time series, computation of initial dot products for each diagonal and the evaluation of the distance matrix entries. The *work* measurement adds to those times the time spent for merging the processes individual matrix profile results into a global one. Namely it captures all required work to produce the global matrix profile, especially it contains the result communication. As explained the *total* timings measure the full algorithm time, including the file input, output and communication.

Strong Scaling

For strong-scaling we use a random-walk time-series of 801 249 samples, computing the matrix profile for 1000 sample subsequences and repeat each measurement 7 times. This corresponds to a 12.8 MB matrix profile, which every process needs to send and receives in the result communication, which is clearly bandwidth bound (see fig. 35). As parallel I/O is



(a) strong scaling

(b) weak scaling

Figure 26 Scaling behavior of trival parallalization: Strong-scaling performed with an 801 249 samples input series and weak-scaling starting from an 89 916 samples input series in the sequential run. The charts show results obtained from manual time measurements. The *comp* graphs are based on the pure computation times, i.e. times for evaluating the distance matrix entries and required precomputations. The *work* measurements additionally include the required result communication to produce the matrix profile. Total algorithmic time, including the file I/O is shown by the *total* data points. We fitted a theoretical model in both experiments independently to the cost behavior.

performed by all processes, the I/O chunks go down to 736 samples, i.e. 5.8 kB for the input, which is quite small. The model of section 5.1.3 can not capture the transition in the domains. Because the observed runtime behavior still shows a good match to the expected one (see the fitted function in fig. 26b), we do not increase the input length: as the runtimes grow with the square of the input length, runtimes would increase dramatically to process sufficiently large inputs to guarantee bandwidth-boudedness of the I/O operations. In our described experiment, parallel runtimes in the experiment range from 1.3×10^3 s down to 4.6 s.

A strong scaling runtime model is obtained from the full theoretical model in equation 5.12 by setting $W = \text{constant}$. The total algorithmic behavior in fig. 26 shows visually good fit to the obtained model with equation:

$$T_{total} = \tilde{c_1} + \tilde{c_2} \cdot p + \tilde{c_3} \cdot p \log_2(p) \tag{6.1}$$

Fitting is performed with SciPys [73] *curve_fit* method, fitting the model function to the means of the experiment (following the approach of Calotoiu *et al.* [59]). The fitting coefficients and error metrics are stated in table 8. While the model gives a good fit, the resulting coefficient $\tilde{c_2}$ is negative, and for this reason the model does not explain a physical behavior, because

the overheads modeled in the function are all positive (see sec. 5.1.3). Possibly a transition from bandwidth to latency bound domain in file I/O causes the mismatch. Alternatively the mismatch is due to unmodeled internal communication behavior in the MPI I/O functions. We will see in the more detailed analysis (figure 27a), that the latter one is the likely candidate.

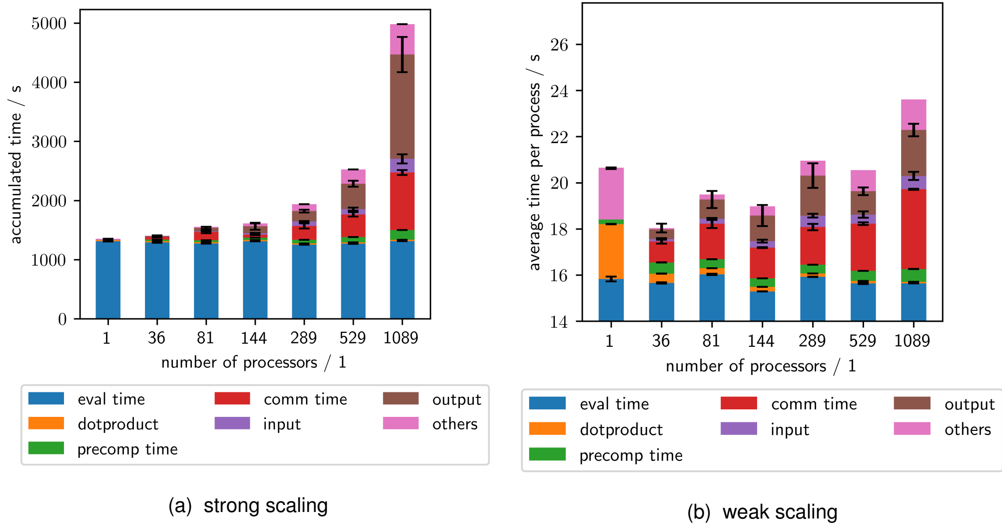
Table 8 Model fit of trivial parallelization behavior: model fitting is performed using SciPys [73] *curve_fit* method, which performs a non-linear fit with a least squares error objective. Fitting data of the strong and weak scaling experiments are costs and parallel runtimes of the total program measurement (see fig. 26b) respectively.

scaling mode	**model ($T_{\mathrm{total,accu}}$ or $T_{\mathrm{total,avg.}}$ resp.)** / s	**RMSE** / s	**adj. R^2** / 1	**SMAPE** / $\%$
strong	$1.4 \times 10^3 - 5.4p + 0.86p\log_2(p)$	89.7	0.995	3.50
weak	$0.69 + 18k + 0.063p^{1.5} + 0.011p^{1.5}\log_2(p)$	544×10^{-3}	0.865	2.34

The scaling behavior of the different program sections in figure 26b indicates, that communication and I/O constitute the largest overheads. Notably with increasing numbers of processes, also the computational part becomes inefficient. A more detailed decomposition of the overheads, as shown in figure 27a explains the behavior: each processes conducts the precomputations for all the input series individually, which is redundant. With increasing numbers of processes, the redunancy overhead becomes noticeable and computational efficiency drops. The redundancy could be avoided by splitting the precomputations among processes and communicating the results, but due to the necessary communication, success is doubtful and obviously the severe bottlenecks are communication and I/O. Among the I/O operations clearly writing constitutes the bottleneck (see figure 27a). In contrast to the idealized theoretical model of section 5.1.3, particularly the output time is not constant but growing. For the very last datapoints, linear growth could be explained by latency boundedness, as the output slices become successively smaller, but it is noticeable for all data points. Potential reasons are internal synchronization or inter-process communication in the MPI I/O implementations. Further investigations like benchmarking of the I/O subsystem are required for a more detailed understanding but omitted here. The implementation is only of minor interest, due to its limitations.

An analysis of the scaling behaviour of individual functions with Extra-P [59], based on application profiles collected with Score-P in the same experiment, suggests that the overhead with worst scaling behavior is file writing. Table 9 lists the functions with dominating growth. As the underlying data contains high variances, the error metrics are quite high and the results hardly reliable. We assume that implicit synchronization in the calls creates variations which are hard to model. Accumulation of the times over different processes further amplifies such variations. In the analysis of the distributed parallelization in section 6.3.2 we will show some plots of such an analysis and discuss the problem in more detail. The only conclusion

we make based on this experiment is that likely the fitted timing model for strong scaling (eqn. 6.1) is underestimating the scaling trend, as the models by Extra-P suggest a faster growth of individual functions, particularly for I/O.



(a) strong scaling

(b) weak scaling

Figure 27 Overheads of the trivial parallelization: based on logged time points in the program, the figure shows the accumulated and average time spent in different code sections in strong and weak-scaling respectively. Be aware that in the weak-scaling most of the evaluation time is invisible due to the adjusted axis range in order to focus on the overheads. The tracked *eval time* is the time spent in the kernel for evaluating distance matrix entries (see alg. 1). Denoted as *dotproduct* is the time to compute the dot products for the first entries in each diagonal (eqn. 4.2) and *precomp time* refers to the precomputation of the meta time-series $\widetilde{\mu_A}$ and s_A (eqn. 4.9). Communicating and merging the local results among all processes (alg. 3) is measured as *comm time*. Time for reading the input data and related communication is denoted as *input*, storing the result as *output* respectively. The remaining time spent in code parts different from the listed ones is shown as *others*. Details of the experimental setup are explained in the respective subsection of section 6.2.

Weak Scaling

For the presented weak-scaling experiment we chose a starting time series length of $n = 89\,916\,\mathrm{samples}$. As the workload is given by the number of matrix entries, we scaled the input length with the square root of the process count in order to maintain a constant amount of work per process (sec. 5.1.3). We compute matrix profiles with a search window length of $1000\,\mathrm{samples}$ and used the same amount of processes as in the experiments for the distributed parallelization (sec. 6.3). While the minimum result message sizes of $\approx n \cdot 16\,\mathrm{B} = 1.4\,\mathrm{MB}$ are reasonably large to saturate the bandwidth, the writing chunks of individual processes range down to $2724\,\mathrm{samples}$ or $43\,\mathrm{kB}$ accordingly. Due to low data amounts, the I/O behavior might possibly be determined by latencies in the experiment. Investigation of the experiment is still of interest: as with increasing numbers of processes, the I/O size in weak scaling decreases (as explained in the next paragraph), the scaling behavior based on latencies constitutes the worst case scaling scenario, as the latencies determine the minimum required time. Also observing a bandwidth dominated scaling behavior for larger input sizes is hardly possible: according to the theory (sec. 5.1.3) and empirically shown later in section 6.4, increasing the problem size per process yields a increase in parallel efficiency and therefore observing the effect of overheads in weak-scaling is delayed towards higher numbers of

processes.

From the theoretical analysis in equation 6.5 we obtain a weak-scaling runtime model by setting $W = p$. As in practice not exact equality but only proportionality is given, the obtained constant factors can not be compared to those of the strong-scaling experiment. Furthermore, with six coefficients and only 7 data points in this experiment, the full model showed significant over-fitting. We empirically set the coefficients c_6 and c_5 to zero, to obtain a reasonable fit. The fitted overall model, as reported in table 8, shows a visually good fit. The bad R^2 fitting indicator is due to the inconsistent variation of the measurements around the model: in some cases the parallel runtime is even decreasing with a increase in the core count in figure 27b. Supposedly the reason is, that the nodes are not completely filled, as we used only square numbers of processes instead of multiples of 28. In combination with the low problem size per node, this causes variations in the caching behavior (available cache amount for each process) and the relation of inter- and intra-process communication. Due to the over-fitting potential of the weak-scaling model, the fitting result should not be relied upon. The only conclusion we obtain from this experiment is, that it shows no obvious contradiction to the empirical results.

A prominent observation in figure 26a is the super linear speedup of the computational program parts. The detailed decomposition in figure 27b shows, that the actual time spent in the evaluation of matrix entries is fairly constant. Variations are likely caused by the partitioning scheme: the number of diagonals assigned to the processes is based on the input length and number of processes and is not related to the chosen block-length of 500 diagonals of the blocked iteration scheme. Therefore the partitioning typically requires processing of incomplete blocks, which causes variations in the throughput (see section 6.1.2). Clearly the decomposition in figure 27b shows that the reason for the super-linear speedup is the drop in the *dotproduct* time, i.e. the time required to compute the dot products of entries in the very first column of the distance matrix. The number of required diagonal initializations grows proportional to the input length. Because the latter is proportional to $\sqrt{p}$ in the weak scaling scenario, but the work is distributed among all p processes without any redundancy, the average initialization work per process is decreasing in weak scaling with $\Theta(\sqrt{p}/p)$, which causes the observed gain in efficiency.

Dominating overhead in weak-scaling appears to be the communication, which coincides with theoretical expectations: the communication is collective over all processes and at the same time the amount of transferred data, i.e. the full matrix profile, is increasing. In I/O on the other hand the size of the written chunks is decreasing, as the input and result length grow only with $\sqrt{p}$. As pointed out, potentially the small sizes cause the I/O behavior to be dominated by latencies which is their worst case scaling scenario. This empirically supports the statement, that the dominating overhead of the trivial parallelization is the result communication.

The Extra-P analysis of the respective dominating functions in table 9, based on the the ap-

plication profiles with Score-P shows similar growth trends for communication as well as I/O. Due to high fitting errors the suggested behavior is not reliable. The high runtime complexities $\in \Omega(p^{2.5})$ suggest, that the theoretical model is underestimating the overheads in the long term and provides only a locally good fit to the data. Particularly this supports the statement of potential unmodeled internal synchronization and communication in the MPI implementation. Notably the time spent in the MPI_File_open call is in the same order of magnitude as the writing operation itself and can't be neglected. This coincides with the observation, that likely I/O behavior is dominated by latency rather than bandwidth due to the shrinking output partitions of the trivial parallelization.

Table 9 Scaling behavior of selected callpaths of the trivial parallelization as reported by Extra-P: data is collected via Score-P instrumentation in the strong and weak scaling experiments described in section 6.2. The analysis considers the time accumulated over all processes. A comparison of instrumentation overhead is given in section 6.4.1

function	**accu. time model** / s	**adj.** $\mathbf{R^2}/1$	**SMAPE** $/\%$
	strong-scaling		
MPI_File_open (result file)	$15.5 + 10.5{\times}10^{-6} \cdot p^{2.5}$	0.998	29.0
MPI_File_write_all	$37.5 + 799{\times}10^{-6} \cdot p^{2}$	0.999	13.8
MPI_Allreduce	$83.0 + 25{\times}10^{-3} \cdot p^{1.5}$	0.979	26.6
	weak scaling		
MPI_File_open (result file)	$6.200 + 19.0{\times}10^{-6} \cdot p^{2.5}$	0.999	58.6
MPI_File_write_all	$24 + 29{\times}10^{-3} \cdot p^{1.5}$	0.969	60.5
MPI_Allreduce	$32 + 10{\times}10^{-3} \cdot p^{1.5} \cdot \log_2(p)$	0.999	43.6

6.3 Distributed Parallelization

6.3.1 Performance Model

We want to start presentation of the distributed implementations scaling behavior with a discussion of performance models of our implementation before analyzing details of the behavior. To collect empirical data, we measure runtime values for every process with manual source code instrumentation, using MPI_Wtime as a timer. We exclude program setup times like MPI initialization and setup of the communicators by synchronizing all processes with a MPI_Barrier before acquisition of the starting time and running the algorithm. The parallel runtime is obtained as the maximum of the logged runtimes of all participating processes.

To build our empirical model, we perform 6 experiments, each consisting of 25 different combinations of input sizes and process numbers. We use 5 different processor configurations, ranging from 81 to 1089 processes. As input time-series, we use random walks (see Chatfield [3]) containing random numbers of embedded motifs with a motif length of 1000 samples, which we also specify as the window-length for subsequence search. Five different input time series are used, the smallest one consists of 801249 samples. The other input lengths are set such that for each chosen number of processes there exists an experiment with the same ratio of distance matrix size to process number as for 81 processes and the smallest input length. Accordingly, the largest series used in the experiment is of length 2.9×10^6 samples. The obtained statistical data are listed in table 22.

The stated input lengths and processor counts are chosen such, that all major messages are sufficiently large, to be in the bandwidth bound domain of the communication network (compare to appended Hpcbench results in 35). For this we ignore the message size of the exclusion reductions. They are relatively small, depending only on the window length and their behavior is masked by the main result reductions (compare section 5.2.3). The smallest relevant message size with 5.8 kB is observed for the slices in input communication in the experiment with 1089 processes and the smallest input series. The largest message occurs at the result reductions of the experiment with 81 processes and the largest 2.9×10^6 samples input series and contains 0.57 MB.

We state models in terms of the cost $T_{\text{total}}(p, k) = p \cdot T_{\text{par}}(p, k)$. The parameter p denotes the number of processes and k the relative number of distance matrix entries compared to the smallest one in the experiment, computed by:

$$k = \frac{\text{work_triangle_length}^2}{(\min\{\text{work_triangle_length in experiment})\}^2} = \frac{(n - w - e + 1)^2}{64 \times 10^{10}} \tag{6.2}$$

where n, w and e denote the input series length, the subsequence search length and the size of the exclusion zone, which we fix to $e = w/4$ in our implementation (see section 3.1.1).

We show the fitting data for two models: the *theoretical* one and an *empirical* one that was suggested during analysis with Extra-P [59]. With coefficients $c_1, c_2, c_3, c_4 \in \mathbb{R}^+$ (which are not shared by the two models) they are given as:

$$T_{\text{total,theo}}(p, k) = c_1 + c_2 \cdot k + c_3 \cdot \log_2(p)\sqrt{p \cdot k} + c_4 \cdot \sqrt{p \cdot k} \tag{6.3}$$

$$T_{\text{total,empir}}(p, k) = c_1 + c_2 \cdot p^2 + c_3 * k \tag{6.4}$$

We fit the computation time models to the medians of the experimental data, following the approach of Calotoiu *et al.* [59], to avoid negative impacts of outliers. A non-linear least-squares fit with SciPys [73] *curve_fit* method yields following models (after initializing all parameters

as 1.0 s):

$$T_{\text{total,theo}}(p,k) = 325.1\,\text{s} + 1287\,\text{s}\cdot k + 8.026\,\text{s}\cdot \log_2(p)\sqrt{p\cdot k} + -73.88\,\text{s}\cdot \sqrt{p\cdot k} \quad (6.5)$$

$$T_{\text{total,empir}}(p,k) = -4.133\,\text{s} + 8.298\times 10^{-4}\,\text{s}\cdot p^2 + 1273\,\text{s} * k \quad (6.6)$$

Measures of the fitting quality are shown in table 10 and detailed timing data is appended in table 23. As our models consist of only up to 4 parameters and the experimental data consist of 25 support points, we argue that the high R^2 metrics in table 10 do not indicate over-fitting but rather good matches of the models to the data. The fitting errors are very similar for both models and do not significantly favor any of them. In disagreement with derivation of the model, the fourth coefficient c_4 of the fitted theoretical model is negative. As typically no negative parallel overheads exist[1], existance of the negative coefficient suggests, that the modeling function does not explain the physical behavior.

Table 10 Goodness of performance model fit for distributed parallelization: Error measures after fitting the different models to the medians of the modeling experiment data in table 22. I.e. 25 data points were used to fit the model. Maximum and minimum of the fitted compute times T_{total} are 2069 s and 8238 s respectively.

error metric	theo. model	emp. model
root mean square error / s	92.11	111.1
R^2 / %	0.9999	0.9998
adjusted R^2 / %	0.9998	0.9998
SMAPE / %	1.071	1.148

In order to validate the predictive power of the models, we perform two large scale test runs. We use random walk time series of 100 005 000 and 6 401 249 samples with three different embedded motifs of lengths 50 000 each. We compute the matrix profiles with 7056 processes, about half an island of the supermuc phase 2 cluster. This is a seven-fold increase of processor number compared to the largest number in the modeling experiment.

The relative errors of the prediction for the large problem sizes in table 11 show a reasonable accurate prediction for both models. Notably both underestimate the overheads, as they show shorter predicted times than measured. For the smaller problem size of the test experiments, both models exhibit quite high prediction errors. As the empirical model is three times as accurate as the theoretical one, it provides the more accurate predictions and should be preferred for such purposes.

[1] in some parallel programs, code parts can actually experience speedup for example due to better cache locality, which could cause shrinking runtimes. Though we know from section 6.1.1 that our kernel does not benefit from decreasing local partition sizes and none of further in depth investigations show such a behavior

Table 11 Large-scale predictions: For two different problem sizes the runtime prediction of the models in equ. 6.6 is tested in a single experiment for each size, using 7056 processes.

	input length $6.40{\times}10^6$ **i.e.** $k = 62.781$		**input length** $100{\times}10^6$ **i.e.** $k = 15607$	
	theo. model	**emp. model**	**theo. model**	**emp. model**
predicted T_{total}/s	$100{\times}10^3$	$121{\times}10^3$	$20.4{\times}10^6$	$19.9{\times}10^6$
predicted T_{par}/s	14.2	17.2	$2.89{\times}10^3$	$2.82{\times}10^3$
measured T_{par}/s	19.0	19.0	$2.92{\times}10^3$	$2.92{\times}10^3$
rel. error $T_{\text{par}}/\%$	28.8	10.0	1.17	3.55

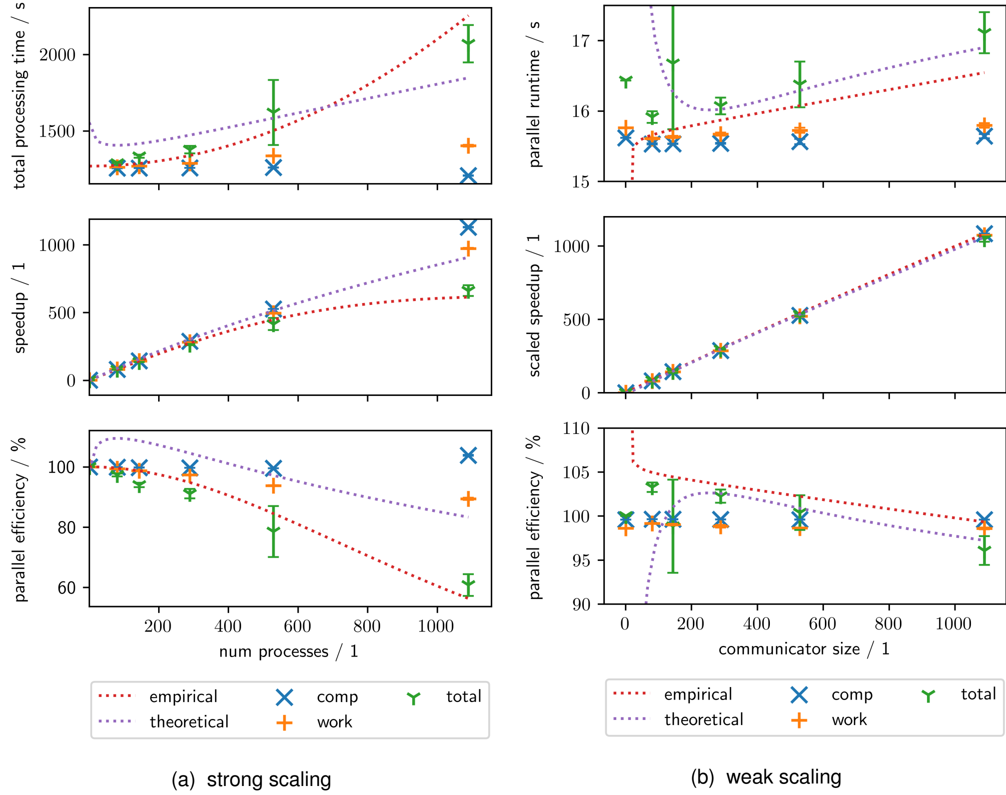
While in both tests, the number of processors is increased by a factor of 7 compared to the largest one in the modeling experiment, the problem size was increased by factors of 5 and 1200. The test results suggest, that the models are quite accurate for predictions with a dominating growth of the problem size, i.e. increasing parallel runtimes compared to the modeling run, as the prediction error is quite low for the large problem. A possible explanation is, that accurate prediction of the workload is dominating and errors in modeling of the overheads are masked by it. In case of the smaller test problem, the increase in the number of processes is similar to that of the problem size. Inaccurate prediction of parallel overheads is observed stronger.

The statement is also supported by the two plotted slices of the modeling data set in figure 28. We selected a subset of the modeling data to show the fitted models in a strong and weak scaling graph. As the modeling experiment does not contain single-process executions, we additionally perform sequential runs to complete the scaling graphs[2]. Among the data of the modeling experiment, the displayed strong-scaling chart shows the subset with the smallest problem size in the data set, to keep the runtime of the additional sequential experiment low.

Compared to runs of executions with bigger problem sizes in the modeling experiment, the selected subset has the lowest cost[3]. Because the least-squares objective of fitting the time models was not weighted, it favors approximately equal errors for all data points. For this reason the shown strong-scaling subset with the smallest input length exhibits the largest fitting errors because of its relatively small absolute runtimes.

[2] I.e. we compute matrix profiles with the parallel implementation and a single process with input sizes of 89 916 samples and 801×10^3 samples for weak an strong-scaling respectively

[3] For example with 1089 processes for the largest problem $T_{\text{total}} = 18.5 \times 10^3$ s according to table 22



(a) strong scaling

(b) weak scaling

Figure 28 Performance models in strong and weak scaling: Strong and weak scaling experiment with the fitted model. Subsets of the modeling experiment data had been used: for strong scaling we used the subset with input lengths of 801 $\times 10^3$ samples. For weak scaling the appropriate series of experiment was selected, starting with the experiment of the same input length using 81 processes. In order to complete the scaling graphs, we recorded additional sequential experiments, i.e. for the weak scaling graph one with a input length of 89 916 samples. The plotted models are the ones stated in equation 6.6, modeling the total algorithm time. The models extrapolated sequential runtime was used in the strong-scaling chart (b) to compute their speedup and efficiencies. As the extrapolation of the sequential runtime for the weak-scaling scenario is rather bad, as seen in the runtime chart of fig. (a), the plotted model speedups and efficiencies are computed based on the additionally conducted sequential experiments. Error bars indicate the standard deviation. With a single In addition to the graph of the total algorithm time, the scaling behaviors based on pure computation time and work time are shown. The comp graphs consider only the time for arithmetic precomputations and evaluation of the distance matrix. The work graphs additionally the time spent for communication of inputs and the result reductions.

The strong scaling plot (fig. 28b) of the total processing time visually clearly shows a better fit of the empirical model. For almost all data points it is within the standard deviation of the data set and notably it also provides a good extrapolation to the sequential run. The long term trend $T_{\text{empirical,ss}} \propto p^2$ of the model on the other hand appears not to be appropriate, as the model seems to increase at a higher rate than the data.

The theoretical model on the other hand shows bad fitting for both, large and small numbers of processes. Such are particularly observed in the shown data subset because of the previously outlined least-squares-fitting behavior. Scale-ups, as proposed by Perlin *et al.* [58] could be used to plot similar scaling charts based on existing data of the modeling experiment without additional long-running sequential executions. As no additional insights for our purposes are expected, we omit it.

The plotted speedups of the models in strong-scaling are computed base on their extrapo-

lated sequential runtime, e.g. $S_{\text{ss,model}}(p) = T_{\text{total}}(1,k)/(T_{\text{total}}(p,k)/p)$. Due to the overestimation of the sequential runtime, the theoretical model shows super linear-speedup, i.e. more than $100\,\%$ efficiency for small numbers of processes. It hardly follows the data trend. In contrast to this, the empirical model shows good coincidence with the data.

In the weak-scaling subset (fig. 28a) of the modeling experiment (with an added sequential run), both models exhibit some weakness. None of them manages proper extrapolation of the sequential run. For this reason we compute the displayed speedup and parallel efficiencies of the models in weak scaling based on the mean of the additionally measured experimental runtimes: $S_{\text{ss,model}}(p) = p \cdot T_{\text{measured}}(1,k_1)/T_{\text{total}}(p,k)$. With these plots we are able to visualize the scaling trend, while pure model based plots would be far off from the experimental and prevent any visual interpretation due to the required rescaling of the vertical axis. Combined with the observation in the strong scaling graph, we propose to compute predictions of the parallel efficiency with the models based on extrapolation in the dimension of the processor number as in the strong-scaling scenario (be aware of the formulation in terms of total processing time, not parallel runtime):

$$E_{\text{model}}(p,k) = \frac{T_{\text{total,model}}(1,k)}{T_{\text{total,model}}(p,k)} \tag{6.7}$$

Looking at the parallel runtimes in weak scaling, the theoretical model appears to provide a better fit towards higher processor numbers. The observed consistently shorter runtimes of the data sets' weak scaling subset are explained by fitting in the 2D input domain. Other data points in the p dimension enforced this model behavior[4]. Notably the more simple empirical model shows a worse fit. Note though that in the long term the predicted runtime of the theoretical model will fall below that of the empirical model due to a slower growth rate: in the weak scaling scenario it holds that $p = k$. Considering the long term trend, with $T_{\text{total,theo}} \in \Theta(p\log(p))$ and $T_{\text{total,theo}} \in \Theta(p^2)$, the empirical model will outgrow the theoretical one and likely show better predictive behavior. The claim is supported by the generally better prediction accuracy in the previously presented test (see table 11).

Comparing the empirical runtime model to the theoretical one, a notable difference is, that it does not contain any mixed terms $T(p,k)$. In combination with the observation of the weak scaling fit, we assume, that the empirical model fits the locally dominating behavior in the modeling test sufficiently well but is a oversimplification: we assume a missing interdependence of the problem size and processor count. We performed further investigations, in order to explain bottlenecks and root causes of the overhead behavior in the implementation, as presented in the next section. As we were still not able to develop a better model, based on the presented results we suggest to use the empirical model for predictions of performance and efficiency in practical applications.

---

[4] In a full 3D plot, one could observe data points below the fitted surface for other processor numbers as in the strong-scaling experiment

6.3.2 Parallelization Bottlenecks

The strong- and weak-scaling graphs in figure 28 give a rough overview of the important overheads: in addition to the *total* computation or parallel runtime, measurements of *work* and *comp*utation times are shown, including speedup and efficiencies based on these times. The total times are based on the maximally measured time among all processes from a synchronization barrier after setting up MPI and parsing the command-line until the results had been written to the file. I.e these times include the file I/O times. The *comp* measurements focus only on the computational times: such are the precomputations of the sliding means, initialization of the dot products and iteration over the distance matrix. Put differently, these times do not track any MPI communication or file I/O calls. The *work* timings add to that the times spent in the communication calls for the result reductions as well as the input broadcasts. Note, that these work times also include some idle time of processes which are waiting to receive input from the I/O-processes. For this reason, the work times are an overestimate of the true time required for producing the matrix profile. We try to separately track those idle times in another experiment, as explained later in this section.

The measurements in figure 28 show a high gap between the total and the work times in contrast to the smaller gap between work and computation times. This shows, that the file input/output operations form the most important bottleneck, at least at the scale of the modeling experiment. Notably the computation times show quite perfect scaling behavior, even in strong-scaling. This shows, that the manual partitioning scheme is well balanced. Notably the computation times also include time of unproductive computations due to the required padding (see sec. 5.2.1) and overheads by the additional initialization of diagonals in the first column of each tile. The nearly perfectly constant computation times match the theoretical expectations, that those overheads are neglectable for reasonably large problems (e.g. evaluation of $64 \times 10^{10}\,\mathtt{entries}$ of the distance matrix in the strong-scaling chart). As the standard deviations show, performance is highly consistent for the computations and even communications.

Strong Scaling

For a more detailed analysis we perform experiments with manual source code instrumentation and measurements with Score-P [60]. In order to keep track of imbalances and important idle-states we add explicit synchronization barriers and track the times spent in those. Barriers are added at points stated in the following list to track the named sections:

idle reading: after reading the input data from file, before the input communication

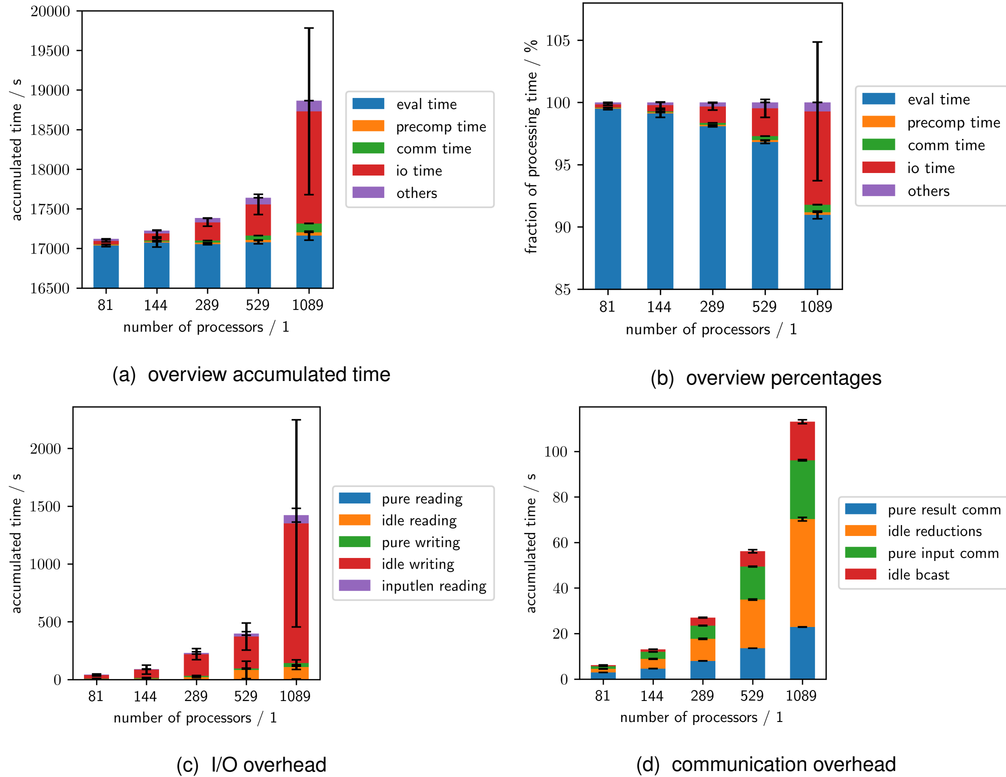
idle bcast: after input communication, before starting the computations

idle eval: after finishing the matrix evaluations, before the result reduction

idle reductions: after finishing the result reduction, before writing to the file

idle writing: after writing the result to the file

Without such barriers, for example the idle times of processes which do not participate in parallel file reading appear as waiting time within the input broadcast call, as all processes are necessarily synchronized to the input ones. We use Score-P compiler and runtime instrumentation [60] in the same experiment to generate profiles of the application. We apply a filter, listed in 8.1, to achieve low overheads, which are demonstrated in section 6.4.1.



(a) overview accumulated time

(b) overview percentages

(c) I/O overhead

(d) communication overhead

Figure 29 Overheads in strong scaling of the distributed parallelization: shown are timings of important program parts obtained with manual source code instrumentation. Added synchronization barriers allow tracking of idle times. Shown values are computed from time-spans accumulated over all processes. (a) gives a overview of the sections in terms of absolute timings, figure (b) shows the same data in terms of percentages. Figures 29c and 29d show a detailed decomposition of the IO and communication times of the same experiment. The attribute "pure" in those charts means, that the measured time was spent in MPI functions after previous synchronization of all processes. This way the timings try to capture mostly the required communication/IO time and exclude waiting times due to synchronization. Time spent in these additional barriers is shown as "idle" times. Measurements were repeated 7 times. The plotted standard deviations refer only to the the timings of the individual bar section at whose tops they are shown, not the entire stacks below. Note that the selected processor counts are distributed approximately exponential. Due to the restriction to square numbers, exact exponential distribution was impossible.

Figure 28 shows results of our source-code time measurements in a strong scaling experiment. As we do not require any sequential run for our further investigations, we computing matrix profiles for 1000 sample subsequences of a 2.93×10^6 sample input series on up to 1089 cores. Those parameters coincide with those of the experiments with the largest size in the modeling experiment. We choose them, because we do not require a sequential run for the further investigations and the message sizes above 703 kB are clearly bandwidth bound. The overview charts 29a and 29b show clearly, that most overhead is generated by file I/O code parts, second are the communication.

Figure 29d shows a further decomposition of the communication fraction. Broadcasting of the input slices composes the smaller fraction compared to the result communication. As the time spent by different processes in the broadcast is roughly the same, little imbalances, i.e. only small *idle bcast* times are observed. With growing number of processes, the accumulated time is increasing as expected. The reductions of the results constitute the dominating communication overhead, as more data compared to the input is to be transferred and the pairwise merging procedure is applied. Implementation of the reduction with a tree communication structure causes some processes to be finished earlier than the receiving ones. For this reason a significant portion of inevitable idle time is observed. While the idle time was measured in this experiment with the help of an additional synchronization barrier, it also shows up as parallel overhead without: Idle times are mostly caused by processes, which do not participate in file output, as those are the ones receiving the reduction results. For this reason the processes idling early during the reduction have no further productive work to perform, as writing the result is the very last step.

Figure 29c shows a more detailed decomposition for I/O operations. The accumulated time spent by the I/O processes only in reading/writing is denoted as *pure reading/writing* time. As a first observation we want to state, that the time spent for reading the input length from the file header and broadcasting it, denoted as *inputlen reading*, appears to be neglectable compared to the other factions. As only $\sqrt{p}$ processes participate in I/O and $p - \sqrt{p}$ are idle, the idle times of reading and writing show the same behavior as the pure input times, magnified by a factor of $(p - \sqrt{p})/\sqrt{(p)}$. Because of this amplification, the idle times constitute the largest fraction. As writing the results is the last step in the algorithm, the respective idle-time can not be filled with any productive work and is observable as overhead also without our instrumentation barriers. The only potential way to reduce it is, to alter implementation such that more processes participate in file output. Similarly for the file input, processes can not start working before receiving input from the respective input processes. For this reason the only option to reduce the idle time would be using more input processes.

In order to check, whether the overheads coincide with the performance model, we parse the presented timings of the program sections into a text file as input for Extra-P [59] and look at the automatically selected empirical performance models. The report for our data is appended in listing 8.2. Generally the report is fairly unreliable, as few models show good fits. The analysis typically lacks accuracy due to high standard deviations. Importantly due to the measurement with barriers, also variations from the barrier communication are tracked in the measurements. While accumulating the times of several processes is useful for interpretation, as in strong scaling the cost in the ideal case stays constant, it also amplifies standard deviations for higher numbers of processes. Automatically selected models for the total accumulated runtime as well as accumulated communication and I/O times in the report are within $\Omega(p^{1.75})$. This seems to correspond with the observation in the strong scaling subset of the modeling experiment, where the empirical model shows a p^2 growth, which appears to exceed the data. Though the fits are quite bad (for example $21\,\%$ SMAPE error for the accu_-

io_time) and therefore not reliable. Similarly with a bad fit (10.5 % SMAPE and 86 % adj. R^2) the accumulated idle times after the matrix evaluation are reported with a $\Theta(p^3)$ behavior. That would imply, that for further increases of the processor number in a strong scaling scenario, those idle times will be dominating. It seems to suggest, that both models of section 6.3.1 provide an underestimate of the true runtime in the long term. A visual plot (appended in figure 36) of the data on the other hand exhibits, that those data give little evidence that the stated model is fitting best and for example a linear trend could also be considered with similar bad fits.

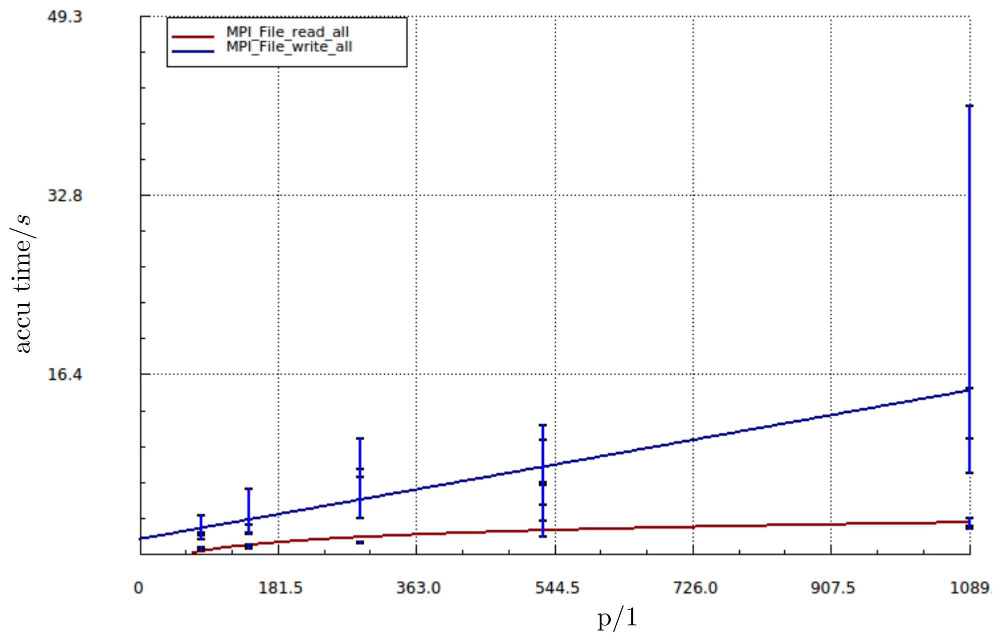


Figure 30 Scaling of I/O functions: Analysis of profiling data from a strong-scaling experiment. Shown is the strong-scaling subset of the modelling experiment with 2.9×10^6 sample input series. Cube profiling data are acquired with Score-P compiler instrumentation and analysis performed in Extra-P. Shown is the result of fitting to the data-means. Table 6.3.2 reports the respective error measures and models. For each data-point Extra-P plots the value range as a bar between maximum, minimum markers. Further the means and medians are indicated by horizontal markers on each of the bars.

To get more accurate and in depth insights, we use Extra-P also to analyze the scaling behavior of individual functions with the set of Cube application profiles, which we collect in the same experimental setting with Score-P [60]. Again high variances, as seen in figure 30 of the data make most of the results unreliable. In addition to the accumulation of runtimes, increased and inconsistent variance for collective MPI calls might be caused by internal synchronization. Fitting either to the means or medians of the data yields different model selections with varying goodness of fit, but none of both options can be preferred in general. Table 6.3.2 shows a selection of functions and their respective analysis results for both scenarios. We selected the MPI_File_read_all and MPI_File_write_all calls among further I/O functions (like opening or setting the views), as they show the dominating behavior. The plot (fig. 30) and tabular data (tab. 6.3.2) suggest that for both important I/O functions assuming at most a linear growth with the number of processes is reasonable. Because only the chosen $\sqrt{p}$ I/O process conduct these I/O calls and all other processes are idling, the total runtime overhead

based on this estimated follows a $\frac{p}{\sqrt{p}} \cdot p = p\sqrt{p}$ behavior. This clearly exceeds the theoretical expectation ($\sqrt{p}$, see tab. 5) and supposedly constitutes the reason for bad fitting and prediction quality of the theoretical performance model. On the other hand, the proposed empirical performance model suggests a p^2 growth and appears to provide a overestimate.

Further analysis of the Score-P profiles shows some functions with higher complexity than p^2. Highest growth supposedly is observed for MPI_File_set_view, as listed in table 6.3.2. As the respective model shows, the fitted growth trend in the data is tiny (note the factor of 7.6×10^{-20}), while again high standard deviations can be observed. For this reason it appears rather to be a over-fitting artifact. Similarly some more functions show bad scalability and at the same time bad fits.

Table 12 Strong scaling behavior of selected functions of the distributed parallelization reported by Extra-P based on Score-P instrumentation in experiments with the distributed parallelization. Using the commandline-interface, fitting was performed against the means and medians of the 7 experiments.

function	**accu. time model** / s	**adj. R^2** / 1	**SMAPE** /%
	mean fitting		
MPI_File_write_all	$1.29 + 0.013 \cdot p$	0.917	18.6
MPI_File_read_all	$-4.30 + 0.707 \cdot \log_2(p)$	0.699	38.8
MPI_Barrier (idle)	$137 + 1.18 \cdot 10^{-3} \cdot p^2$	0.995	13.7
eval_tile_blocked	$1.70 \cdot 10^4 + 9.46 \cdot 10^{-3} \cdot p$	0.786	7.10 $\times 10^{-3}$
MPI_File_set_view	$2.27 + 3.98 \cdot 10^{-16} \cdot p^4 \log_2^2 p$	0.999	3.23
	median fitting		
MPI_File_write_all	$-13.9 + 2.387 \cdot \log_2(p)$	0.806	26.9
MPI_File_read_all	$-0.096 + 7.51 \cdot 10^{-3} \cdot (p^{0.5}) \log_2(p)$	0.996	4.88
MPI_Barrier (idle)	$76.0 + 9.40 \cdot 10^{-3} \cdot p \log_2^2(p)$	0.998	8.02
eval_tile_blocked	$1.70 \cdot 10^4 + 0.147 \cdot \log_2^2(p)$	0.868	5.67 $\times 10^{-3}$
MPI_File_set_view	$0.027 + 7.76 \cdot 10^{-20} \cdot p^{-5} \log_2^2(p)$	0.271	16.3

Overall, we find no evidence in this experiment to disprove the p^2 growth of the empirical performance model, but also do not have a clear proof that it describes the actual physi-

cal behavior. We want to note, that visually the minima of the runtimes for some functions seemed to provide a more consistent behavior, but fitting to such is not available in Extra-P. Similar to optimal computation times [74], runtimes for functions involving communication are lower bound by the highest achievable bandwidth, lowest latency and zero synchronization overhead. This makes the minima potentially more consistent and interesting for modeling the scaling behavior and might be a interesting extension to the Extra-P utility.

Weak Scaling

For analysis of the weak scaling behavior we perform a experiment using a search window length of $1000\,\text{samples}$ and for the sequential run a input length of 89916. Scaling is performed from 1 to $1089\,\text{processes}$ (1 to $39\,\text{nodes}$ respectively), following approximately a exponential schedule[5]. The input length is scaled with $\sqrt{p}$, such that the relative problem size compared to the sequential run (see equ. 6.2) equals the process number.

Analysis of the weak-scaling behavior of the distributed parallelization adds no further insights to the strong scaling analysis. As figure 31 shows, again the dominating overhead at the scale of the modeling experiment is clearly the File I/O. Note that the figure shows the average time per process, which should stay constant in an ideal weak-scaling experiment. For this reason the plotted I/O time shows approximately the actual I/O time of the chosen I/O processes or respectively the idling time of idle ones (for partitioning see fig. 21). As again $\sqrt{p}$ processes perform the actual I/O work, while $p - \sqrt{p}$ are idle, similar to the strong scaling experiment most of the I/O overhead is composed of idling time. Similarly variations, i.e. the standard deviation, of the I/O times are amplified by the idle processes. As not all processes perform the same task, plotting the average time among all processes creates a misleading image when performing a decomposition into idling and active processes, for this reason we omit detailed plots similar to those of the strong scaling experiment.

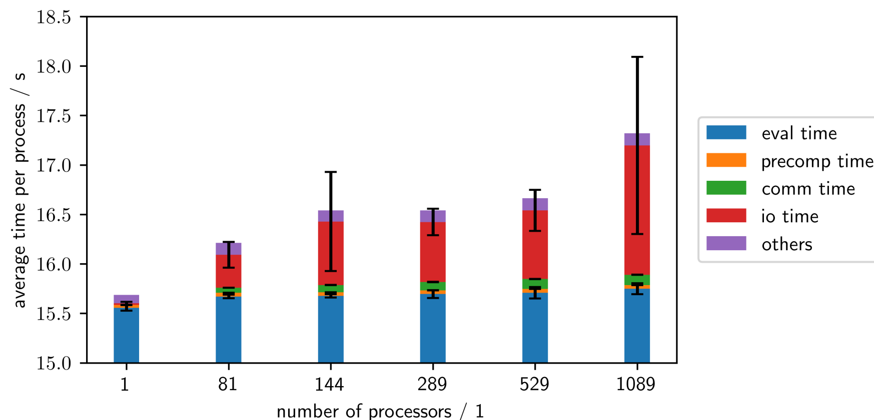


Figure 31 Overheads of the distributed parallalization in weak scaling: show is the average time per process, which is computed from measured times accumulated from logs of all processes. The denoted measurment sections are the same as in figure 29

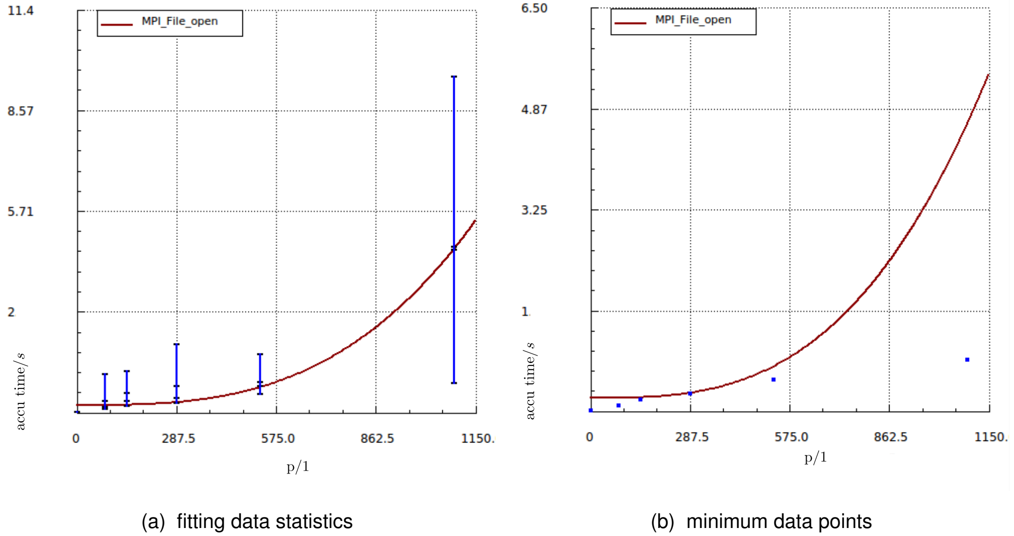
[5] the implementation restricts us to square numbers

We also examine the scaling behavior of custom measurements as well as individual functions with Extra-P. Unlike Calotoiu *et al.* [59] we experienced analysis of the weak scaling behavior not to be easier or more accurate than the strong-scaling analysis. We analyze accumulate rather than parallel runtimes, especially during the exploration of the CUBE profiles recorded with Score-P for the following reason: analysis of parallel runtimes of functions in which not all processes participate, such as MPI_write_all in our implementation, does not give meaningful results: non-participating processes are accounted with runtimes of $0.0\,\mathrm{s}$, which distorts e.g. the means and finally the selected models. Relying on accumulated runtimes as in strong-scaling also accumulates variations like synchronization times and errors. For this reason the fitting data again shows tremendous variances and a decision upon a correct model is quite impossible.

We append a selection of the Extra-P fitting experiment in table 24. Note that the listed models of the I/O functions accumulate only the times of the I/O processes. As at the same time the remaining processes are idle, as outlined in the preceding strong-scaling discussion, models for the overall runtime impact should can be derived by adding a factor of $\sqrt{p}$ to the behavior. For this reason, the total I/O behavior which results from the suggested models for the actual reading and writing calls is between $\Theta(\sqrt{p}p))$ and $\Theta(\sqrt{p}\log_2 p)$. The latter one roughly coincides with the p^2 growth of the empirical performance model, taking into account the hard empirical distinction between $\sqrt{p}$ and $\log_2(p)$ behavior [59].

The measured idle times of the synchronization barriers are aggregated in a single MPI_Barrier call in the Extra-P analysis, as their call-stacks are identical. I.e. the I/O idle-time, which comprises the largest overhead, is contained in there. The reported scaling behavior of $\Omega(p^2)$ shows a high modeling error, due to high variance in the data. The proposed empirical performance model with its p^2 overhead term appears to follow the trend of this overhead portion.

The dominant long-term behavior reported in our Score-P profile based analysis is observed for the MPI_File_open calls, i.e. for opening the result output file. The reported behavior $\Theta(p^3))$ is not captured in any of the performance models. Due to the small factors the behavior becomes noticeable at more than 100×10^3 processes. Fitting errors of the suggested models are quite high, arguably due to internal synchronization or varying communication overhead within the functions. Figure 32 shows fitting of the accumulated time over all processes. We plotted only the minima of the data points in figure 32b, to which fitting is not available in Extra-P. Those appear to follow a more moderate $\Omega(p)$ trend, maybe even $\Omega(\sqrt{p})$. As explained in the context of the strong-scaling bottleneck analysis (6.3.2), arguably the minima are better suited to explain the scaling behavior. For example the $\Omega(\sqrt{p})$ behavior would match a theoretical explanation of constant inclusive function time per processes, as the implementation uses $\sqrt{p}$ I/O processes. For this reason we suspect that the fitting data is highly distorted, e.g. by variations due to communication times and the call does not truly expose a scalability bottleneck in the long term.



(a) fitting data statistics

(b) minimum data points

Figure 32 Scaling behaviour of MPI_File_Open in weak-scaling: The call is reported as a long-term dominating model in the analysis with Extra-P. (a) shows the statistics of all data points, which are time spent in the function accumulated over all processes: the vertical delimiters illustrate minima and maxima, the remaining markers the mean and medians of the fit. (b) shows the same model, plotting only the minimum data points.

As in the analysis of strong scaling bottlenecks, we do not find a clear indication to reject the empirical performance model of section 6.3.1 in the long term. On the other hand the insight provide only little support due to high fitting errors.

6.4 Comparison of Implementations

In the previous sections we separately discussed and examined two parallelization approaches and their scaling behavior. We investigated the major bottlenecks at the examined scale and verified, that no long-term dominating bottlenecks are hidden. In section 6.4.1 now we provide a direct comparison between the implementations to show the gains of the distributed version. By comparison to a version with the applied profiling and instrumentation techniques, we further validate the previously gained insights. We conclude our experiments with a empirical comparison to the SCAMP framework, which became available concurrent to our work (see also sec. 4.4).

6.4.1 Trivial and Distributed Parallelization

According to the theoretical performance analysis in sections 5.1.3 and 5.2.3, the isoefficiency metrics of the presented trivial and distributed parallelization are $\Theta(p^2 \log^2(p))$ and $\Theta(p \log^2(p))$ respectively. It states, that the dominating long-term growth of overheads is lower for the distributed parallelization. For this reason, in theory the distributed parallelization is expected to show better scalability. Root cause of the better isoefficiency metric are the comparatively smaller message sizes in the distributed parallelization due to the tile partitioning in contrast to the fully transferred input and result arrays in the trivial one. It is revealed when comparing the individual terms in tables 5 and 4.

The isoefficiency metric is concerned with the modeled log-term dominating overhead, but we observed deviations from the theoretically modeled behaviors in the scaling experiments (sections 6.2 and 6.3). Figure 33 puts the previous strong scaling experiments of the implementations next to each other. For the distributed parallelization we show the un-instrumented version from the modeling experiment (see figure 28b) as well as a version with Score-P instrumentation and barrier synchronization, to justify our insights based on the low overhead of the instrumentation.

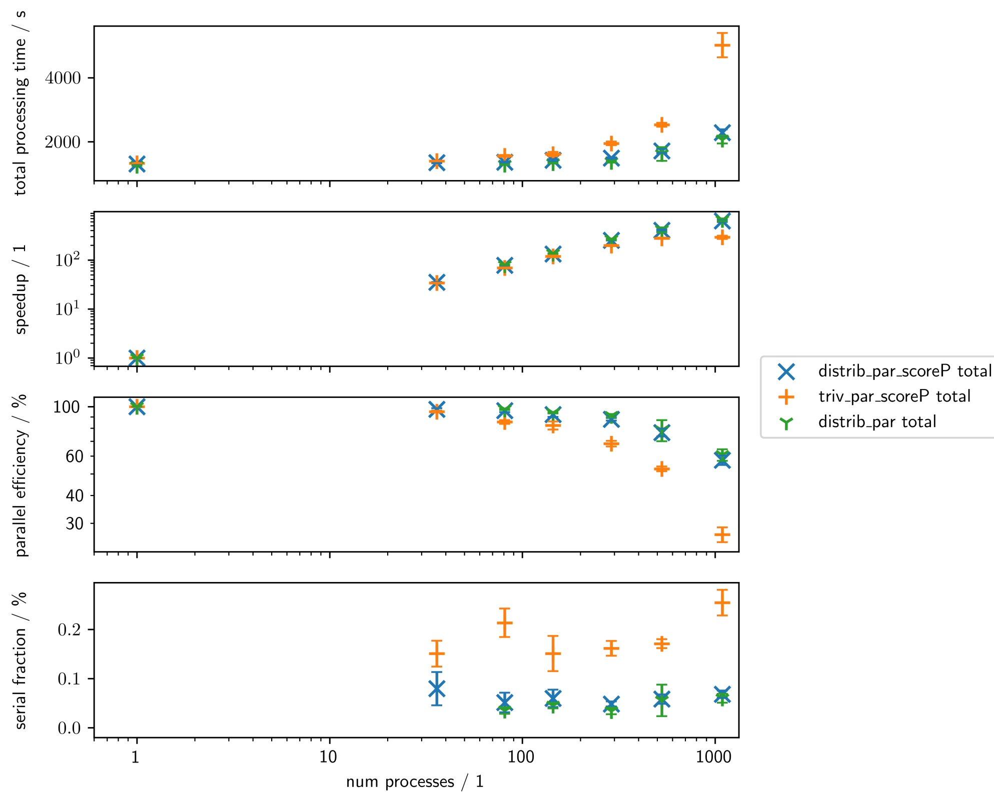


Figure 33 Comparision of implementations in strong scaling: The running example of a strong scaling experiment with a 8×10^5 sample input series and 1×10^3 sample window-length (see fig. 26,28b) is used for a direct comparison of implementations. To justify instrumentation overhead, results for the distributed parallelization are shown with and without Score-P instrumentation. The shown total runtimes include the I/O time. Interested readers can compare the diagrams of the respective subsections for more details. Mean times, relative instrumentation overhead and implementation speedup are appended in table 26.

The figure clearly shows the superiority of the distributed parallelization. In the sequential experiment, the runtimes are indiscernible, as the amount of work is the same and no inter-process communication or parallel I/O involved. For the maximum number of 1089 processes, the distributed parallelization is more than two times faster than the trivial version due to lower overhead (see also tab. 26). In particular the consistently lower serial fraction by almost one order of magnitude visualizes the superior scaling behavior. In addition to the better scaling behavior, the problem size for the trivial parallelization is limited by the single-node hardware as explained in section 5.1.1. Summed up, the distributed version is clearly preferable for usage on a compute cluster, as it provides better scalability.

Figure 33 furthermore validates our bottleneck investigations based on a program version instrumented with Score-P and a few additional synchronization barriers (see section 6.3.2): little runtime overhead is observable when instrumentation is applied and the scaling behavior is unmodified. The relative runtime overhead based on the cost grows to a maximum of $10\,\%$ for the highest number of processes(see table 26), in which the parallel runtime is already as small as $2\,\text{s}$.

6.4.2 Comparison to SCAMP

In order to compare our implementations, we repeat the experiment reported in the publication of SCAMP [2]: for time series lengths from 2^{18} up to 2^{23} samples we compute the matrix profile with a fixed number of processors. Zimmerman *et al.* [2] used 72 OpenMP threads in their shared memory implementation, which is the maximum number supported on their hardware[6]. As our distributed implementation requires a square number of processes, we choose to execute our implementations with close-by 81 processes on 3 nodes. A comparison based on the absolute runtimes is therefore pointless. Instead we consider the average computational throughput per process, based on the parallel runtime:

$$\text{avg_throughput} = \frac{k}{p \cdot T_{\text{par}}} \tag{6.8}$$

where p denotes the used number of processes, i.e. 72 and 81 respectively. We model the parallel program cost depending on the problem size as:

$$T_{\text{total}}(k) = p \cdot T_{\text{par}}(k) = c_1 + k \cdot c_2 + \sqrt{k} \cdot c_3 \tag{6.9}$$

For fitting to the data we use only a approximation of the problem size $k \approx \text{input_len}^2$. The model matches the theoretical models of the trivial and distributed parallelization for a fixed number of processes (see eqn. 5.4 and 6.5). The term $c_2 \cdot k$ models the runtime required for processing the workload, the remaining ones the overhead. Communication overheads depending on the number of processes are subsumed in the constants, as the process numbers are fixed. While we perform no in-depth analysis of the SCAMP implementation, the SCAMP algorithm involves operations on the input and output time series of length $\sqrt{k}$, like precomputation of meta time-series similar to ours. This justifies applicability of the model, despite the reasonably good empirical fit: SMAPE metrics are below $1\,\%$, the RMS errors below $1\,\text{s}$, as listed in detail in the appended table 25. The obtained models are listed in table 13, together with the maximum speedup derived from the models.

The plot of the data in figure 34 shows, how the average throughput increases with the problem size and saturates at some point. Obviously, the growth of the parallel runtime in eqn. 6.9 is dominated by the work-dependent term. Parallel overheads become neglectable with

[6] reportedly they used a c5d.18xlarge instance on the Amazon AWS cloud. The respective CPUs are Intel Xeon Platinum 8000 series, supporting AVX-512 clocked with 3.0 GHz up to 3.5 GHz with TurboBoost. Notably they provide up to 72 hyper-threads, therefore only 36 physical cores, according to https://aws.amazon.com/de/blogs/aws/ec2-instance-update-c5-instances-with-local-nvme-storage-c5d/ and https://aws.amazon.com/de/ec2/instance-types/

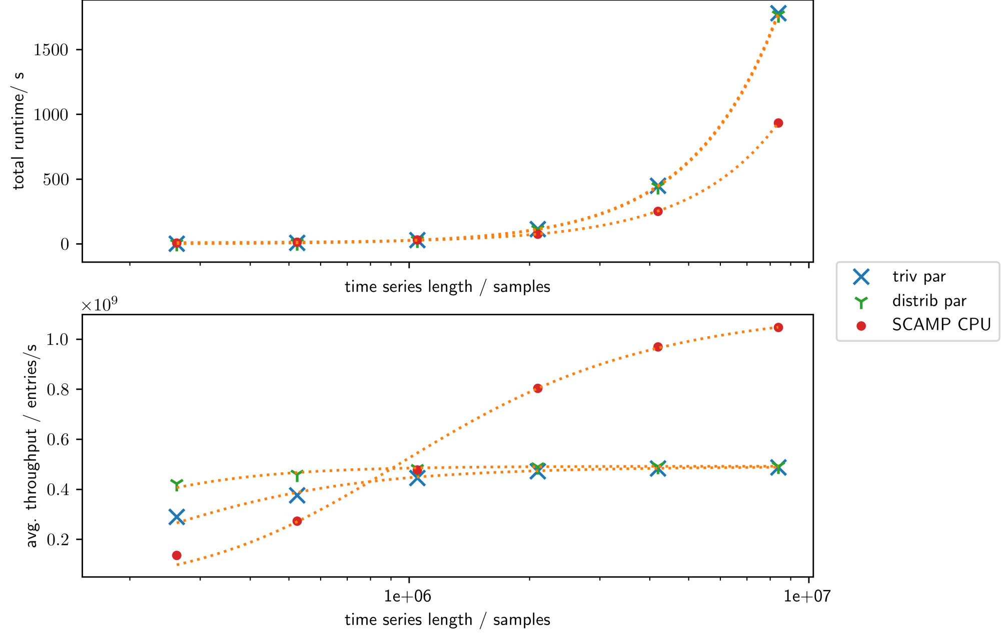


Figure 34 Throughput comparision, including SCAMP: Plotted timings for SCAMP are those reported by Zimmerman *et al.* [2] for its OpenMP CPU implementation utilizing 72 threads. We ran our implementations on the same problem sizes using 81 processes, due to the restriction to square numbers of our distributed parallelization. For each input size, we compute the matrix profiles with subsequence window lengths of 200, 1000 and 5000 samples respectively, repeating each measurement 2 times. We omit to plot standard deviations, as they would be hardly recognizable. To compare the implementations, we computed the average throughput per core according to equation 6.8.

sufficiently large problem sizes. For this reason the throughput towards large problem sizes approaches the maximum throughput achievable by the computational kernel. This implies, that also in the limit all the communication and I/O overheads diminish towards a zero fraction of the overall time. While Zimmerman *et al.* [2] did not include I/O overheads or merging of partial results in their measurements, this property enables a comparison of the results.

We derive the maximum achievable throughput per thread of each implementation from the fitted models as:

$$\lim_{k \to \infty} \text{avg_throughput} = \lim_{k \to \infty} \frac{k}{p \cdot T_{\text{par}}(k)} = 1/c_2 \tag{6.10}$$

Based on these maximum throughputs, we compute scale-ups of the different implementations relative to our distributed parallelization. All values are listed in table 13.

Both of our implementations show in the limit the same maximum throughput. This was to be expected, as both utilize the same kernel. The throughput measured for the used intrinsics kernel in the sequential experiments (see fig. 25b) coincides with the one observed here. Though the effort of the distributed implementation is justified by the fact, that the trivial parallelization is limited in the maximum problem size by the available DRAM, as explained in section 5.1, independent of the number of used processes. The derived theoretical limit of a 52×10^6 samples $\approx 2^{26}$ samples input series is only a factor of 8 from the maximum one used

in the presented experiment.

The CPU SCAMP implementation of Zimmerman *et al.* [2] appears to be more than two times faster than our kernel. The comparison is biased though, as their reported data stems from more modern and more performant hardware. As no peak performance or the exact CPU model are known, it is impossible to provide a accurate comparison. From the available hardware description[7] it can be extracted, that the hardware of their publication provides AVX-512 in contrast to ours (see table 15), which possibly provides usage of SIMD vectorization with twice the vector length of our AVX-2 machine. Further the clock speed on the AWS cloud with 3.0 GHz is significantly higher than the 1.8 GHz available on our system at the time of the experiments (see appendix 8.1) providing further potential speedup. Those data suggest a very approximate hardware based scaleup capacity of 3.3. From the theoretical comparison in section 4.4 a further hardware independent speedup of up to 1.25 based on the optimized arithmetic is to be expected. Both effects combined, it is expected to see a speedup of 4.1 in the experimental data. It is important to note, that the used 72 threads in the SCAMP publication correspond to hyper-threads. Possibly this choice was made to overcome a cache latency bottleneck. It implies that the average throughput per physical core of the CPU SCAMP actually provides a scaleup of 4.6 compared to our implementation, which exceeds the expectation. The results of our kernel analysis in section 6.1.1 on the other hand showed, that our kernel implementation is not fully efficient and could possibly be improved on the current hardware by a factor of 3. We can only conclude based on these data, that both implementations kernels are close to their respective computational hardware limitations and differences in performance for large problem sizes are experienced due to different vectorization efficiencies. Based on the theoretical analysis (sec. 4.4), the SCAMP kernel is superior. Interested readers need to rerun the application on a common target machine to determine exact speedups.

[7] reportedly they used a c5d.18xlarge instance on the Amazon AWS cloud. The respective CPUs are Intel Xeon Platinum 8000 series, supporting AVX-512 clocked with 3.0 GHz up to 3.5 GHz with TurboBoost. https://aws.amazon.com/de/blogs/aws/ec2-instance-update-c5-instances-with-local-nvme-storage-c5d/ and https://aws.amazon.com/de/ec2/instance-types/

Table 13 Model-based comparison of different implementations: we model the parallel runtimes of different implementations depending on the problem size $k \approx \mathrm{input_len}^2$ for a fixed number of processes. Based on the models for a theoretically infinite input series, the average throughput per process is computed. To compare the models, the scaleup relative to our distributed parallelization is shown based on these maximally achievable throughputs. Error metrics for the fitting quality of each model are appended in table 25 due to the lack of space. Fitting quality is generally really good, with SMAPE errors below $1\,\%$ and RMSE errors below 2.1 s, while the data set contains parallel runtimes up to 1.7×10^3 s. Note that the throughput of the GPU SCAMP implementation is computed according to eqn. 6.8 with $p = 1$ and for this reason benefits from the GPU parallelism in comparison to the single-process throughput of the CPU implementations.

impl.	**par. time model** /s	**max throu.** /entries/s	**scaleup** /1
trivial par.	$1.1 + k \cdot 2.5\times10^{-11} + \sqrt{k} \cdot 1.5\times10^{-6}$	490×10^6	1.0
distrib. par	$3.5\times10^{-1} + k \cdot 2.5\times10^{-11} + \sqrt{k} \cdot 3.6\times10^{-8}$	490×10^6	1.0
SCAMP CPU	$7.0 + k \cdot 1.2\times10^{-11} + \sqrt{k} \cdot 7.1\times10^{-6}$	1.1×10^9	2.3
SCAMP GPU	$-9.4\times10^{-2} + k \cdot 1.3\times10^{-12} + \sqrt{k} \cdot 7.4\times10^{-7}$	780×10^9	1600

A look at SCAMPs CPU kernel code reveals, that Zimmerman *et al.* [2] managed to employ auto-vectorization for their kernel. This provides the advantage, that it automatically adapts to the provided hardware capabilities, while our intrinsics kernel requires manual reworking. This constitutes a further advantage of their implementation.

The preprint of SCAMP [2] reports the CPU data only for comparison. Discussed and highlighted is specifically the variant using GPU compute nodes, which achieve the highest throughput. Comparing the saturating throughputs of running SCAMP on one Nvidia V100 GPU to our saturated single-core throughput in table 13 we observe, that 1600 processes or 58 nodes are required on our system in order to achieve overall the same speed. The result of the CPU/GPU comparison is similar to and supports the conclusion of Zimmerman *et al.* [2]: while the economic cost of GPU nodes is typically higher than that of CPU nodes, their tremendously superior throughput makes them economically more efficient for the task at hand. The matrix profile computation is particularly suited for GPU computation, as it is mostly a numerically intensive computation and most of the time does not not require communication.

7. Discussion

In this thesis we presented in depth investigations to improve the performance and scalability of the matrix profile computation, in order to enable previously intractable analysis of time series with almost unlimited detail or length. In particular we persented sequential optimizations to the kernel to speed up the computation, provided a trivial parallelization similar to the suggestions of Zhu *et al.* [32] as a baseline, discussing its limitations and and proposed a distributed parallelization to resolve them. After in-depth experimental analysis and theoretical comparisons of the implementations, we want to summarize our gained insights, critically discuss our contributions and results and hint at potential improvements or future work.

Namely we want to discuss and share our experiences of using MPI for our implementations in section 7.1. We discuss insights and statements about the scalability of the approaches gained from our experiments in section 7.2 before we discuss the presented performance model in section 7.3. We contrast our contributions against the SCAMP framework in section 7.4 and critically discuss weaknesses and validity of the approaches, before we conclude our work with a short summary and outlook.

7.1 MPI Implementation Choices

In sections 5.1.2 and 5.2.1 we outlined the utilization of MPI in our implementations. We want to critically discuss major choices in our implementation and their impacts to share our experiences and hint at potential points of further investigation.

Result communication

Both of our implementations employ a user defined reduction operation to communicate and merge individual processes intermediate matrix profiles. As explained, the custom MPI datatype is defined as a structure of arrays to match the memory layout of the slices used during computation. As a result the reduction is always invoked with a length parameter of 1. While the MPI implementations are allowed to internally split the data to optimize the data transfer, the reduction operation is forced to processes the full transferred data in one pass [67, p. 184], which might have a negative performance impact.

As an alternative, the intermediate computation results could be copied to an array-of-structures before the result reduction. While this increases the memory consumption due to an additionally required buffer, overall there is little runtime overhead generated: because we finally utilized exactly such a data structure for the binary file writing, this conversion is already performed in our current implementations by all I/O processes before writing and the change would only shift its location (also adding it to all processes non-I/O processes). The change

might yield some additional speedup of the result communications. As it is dominated by the I/O fraction in both scaling scenarios of our distributed parallelization, we consider it only a minor problem.

The outlined change still implies copying of the result data for the sole purpose of generating a communication-friendly datatype. The MPI standard designed derived datatypes specifically with the intention to "allow one to transfer directly, without copying, objects of various shapes and sizes"[67, p. 83] and let the MPI implementation decide upon the optimum transfer mode. Under the assumption, that we did not miss a possibility to map the structure-of-arrays matrix profile to a derived datatype with variable length instead of the length 1 data-structure, it could be possibly interesting to consider a modification of the MPI standard: the structure-of-arrays pattern is often used due to optimized memory access and transfer of such data might be performed in more implementations. Because changes of the MPI datatype mechanism could have major implications on all the various communication primitives, we leave the discussion to potential future work.

File output pattern

Both investigated implementations share a common output pattern: results are merged with a reduction operation before parallel writing of the file. While the trivial parallelization uses all processes, the distributed implementation relies only on a subset to store the results. In both implementations only small modifications are required to modify the chosen number of processes, for example one could utilize all processes in the distributed parallelization after switching to a *All-Reduce* or *Reduce-Scatter* result communication.

For both implementations, empirically the I/O constitutes the major strong-scaling bottleneck. This raises the question, which approach is to be preferred or more generally which number of processes is to be used for such a output pattern to achieve overall lowest overhead? The same access pattern appears to be known also from different algorithms, as the MPI standard explicitly states to have opted against implementation of file access based on patterns like reductions [67, p. 491]. Therefore the question applies more generally than just to our case of the matrix profile.

A direct comparison of the overheads of both our implemented output approaches is unfair, as the communication in the trivial parallelization involves the full matrix profile in contrast to the smaller sections of the tiling scheme. For a more general statement, one should also take into consideration access patterns different from ours and consider capabilities of the underlying parallel file system, as they impact the scaling behavior [68]. Due to the required effort, we leave such investigations to future work.

File I/O limitations

As explained by Gropp *et al.* [75, chapter 8], handling of data-structures with a number of elements beyond the integer limit within the MPI requires usage of derived data-types which partition large data into sub-arrays

For the trivial parallelization, this fact has no implications, as the limitation by available main-memory is more restrictive. In case of our distributed implementation, the communication of time-series and profile slices based on arrays of primitive datatypes limits the maximum tile base length to the $2 \times 10^9\,\mathsf{samples}$. Limiting the tile size limits only the maximum amount per partition arbitrary larger series can be processed with more compute nodes with the partitioning scheme. Further this maximum tile size exceeds the maximum overall problem size reported in the literature [2] and far off from tile sizes with reasonable compute times.

Limiting in our implementation though is, that the processes individual file views for writing the result and reading the input series are based on such simple arrays, too. As MPI_Type_create_subarray requires specification of the overall array length, currently the I/O implementation limits the maximum input and matrix profile length in our implementation to the maximum integer value of 2^{31} samples. While such a large problem is still beyond the maximum demonstrated use case by a factor of 2, potential users should be aware of the limitation. It does not in general limit the approach as a few minor modifications (outlined by Gropp *et al.* [75, pp. 243 sqq.]) can be applied to the I/O code.

7.2 Scalability of the Approach

As elaborated in section 5.1.1, straight-forward usage of the SCRIMP parallelization proposed by Zhu *et al.* [32] is limited in the maximum problem size by the available memory at a single node, in case of our hardware to a theoretical maximum of a $52 \times 10^6\,\mathsf{sample}$ time series.

The proposed and implemented tile-based scheme distributes the input series across a compute cluster such that this hardware limit only applies to the single-node problem size. We assessed the quality of our parallelization in strong and weak scaling experiments. In strong-scaling clearly the empirically dominating bottleneck is file I/O. In weak-scaling communication takes a similar share and it is not entirely obvious, which will be dominating in the long-term. Most of the I/O time is composed of idle time, as only a subset of the processes is participating in the collective I/O operations and we pointed out, that future work could possibly further optimize that part.

We argue, that in practice our implementation is fully sufficient and provides a high quality of parallelization: based on the investigations in our work, we chose the size of the scaling experiments to the best of our knowledge, in order to investigate the scaling-behavior and bottlenecks. While in the strong scaling experiment the parallel efficiency dropped towards

30 % at 1089 processes, the parallel runtime already went down to 1.9 s (see table 22). While I/O times are increasing in the scaling scenario, the efficiency is mostly dropping in the experiment due to the fact, that the parallel runtime with its ideal $1/x$ behavior approaches them due to decreasing amounts of work per node. Similarly in weak scaling the parallel the parallel runtime starting at 15.0 s is quite low because the according problem size was selected in order to show the overhead behavior. Such short runtimes are dominated by the time to schedule such computations at the HPC cluster and users would rather use fewer nodes with longer computation times in order to get faster response due to lower resource requirements. Importantly, longer parallel runtimes correspond to larger workloads per process. Those in turn, as demonstrated in section 6.4.2 yield increased efficiencies. This also is a reason, why the large-scale performance model tests of 6.3.1 achieve highest parallel efficiencies (see tab. 23).

7.3 Performance Model

The saturation of the kernel performance with increasing problem sizes per node also is the main reason of applicability of the performance models in section 6.3.1. We observed a negative fitting coefficient for a overhead term of the theoretical performance model, which is physically pointless. The empirical one on the other hand showed visually bad fitting trends in the scaling behavior, supposedly due to the lack of a input length dependent over-head term. In the prediction experiment with the rather small 6.4×10^6 samples time-series, both showed a undesirably high error. For the large problem with a 1.0×10^8 sample input, both show a acceptably small error. The reason is that in the latter case growth of the per-process-workload and parallel runtime compared to the modeling run is dominating the increase in the number of processors and according runtimes: For this reason inaccuracies in modeling of the overheads are out-weighted by the accurate problem-dependent modeling and the relative error ends up small.

Based on that observation, a very simple model, assuming the saturated throughput of the kernel, provides similar competitive accuracy in practice. It can be obtained by fitting to a experiment with varying input size as in table 13 or a single experiment with a saturated node as presented by Zimmerman *et al.* [2] for SCAMP. Both variants at the same time require fewer experimental data and effort than our modeling experiments with Extra-P or manual fitting.

In defense of Extra-P it is to be noted, that it served us well to detect scalability bugs in our development process, which is its primary intention [59]. As we proposed in the bottleneck analysis, a potentially useful extension might be the option to employ the minimum timings in a experiment for modeling: communication operations and MPI collectives are hard to model, as the timings potentially contain high variations caused by wait-states e.g. by synchronization.. As zero-second wait states constitute a lower bound, the minimum timing can potentially capture scaling trends better, similar to the optimum CPU times in micro benchmarks

discussed by Lemire [74].

For more accurate performance prediction, dedicated benchmarking approaches, as proposed for example by [76], are likely better suited. User interested primarily in low runtimes at the cost of lower efficiencies might need to conduct further investigations in that direction. For typical users who are interested in a configuration with high parallel efficiency, the provided models still suffice: they show reasonable accuracy for applications where runtime is dominated by the workload, which is exactly the high-efficiency domain. Users can utilize the empirical performance (which showed highest accuracy in the prediction test) and compute a estimate of the efficiency base on equation 6.7. For high efficiencies, e.g. above $99\,\%$, which are easy to achieve due to the algorithms benign behavior (see. section 6.4.2), the prediction is reliable.

7.4 Comparison to SCAMP

Concurrent to our work Zimmerman *et al.* [2] presented their cluster parallelization, employing a work-queue based batch processing at a AWS cluster.

Similar to our work, as detailed in section 4.4 they employ a blocked iteration scheme and modified arithmetic, which in theory provide further runtime improvements up to $20\,\%$. Because they furthermore conducted experiments, which demonstrate improved numerical stability, we advise to adapt their kernel, which we omitted due to our time constraints. More important, our comparison in section 5.2.4 supports the claimed superiority of their chosen GPU acceleration (also economically). While the queue based batch processing approach causes many disk accesses for storage of intermediate results and few information regarding the overheads are provided in the publication, the scaling behavior of the algorithm justifies the choice: the workload and related runtime increases with the square of the input length but the overhead is only proportional to it, as demonstrated in section 6.4.2. For this reason the overhead is vanishing with increasingly larger input series, which constitute the application domain of the cluster version.

Usage of disk storage in the SCAMP framework on the other hand offers additional advantages [2] compared to our implementation: the computation is preemptible and tolerant to faults of single nodes. In particular the queue based approach makes the computation malleable: the number of used processing elements can be easily adapted to the available cluster resources.

The manifold advantages of the simple disk-based approach of Zimmerman *et al.* [2] raise the question, whether the choice of the MPI as a parallel framework for the task at hand was the best choice. The particular properties, which enable efficient application of the SCAMP approach, is the low amount of required inter-process communication (with $\mathcal{O}(n)$ dependency on the input) in conjunction with the $\mathcal{O}(n^2)$ runtime behavior of the matrix profile computation.

MPI on the other hand is focused on providing efficient communication for large clusters. In particular, the SCAMP framework also benefits from the high GPU throughput, which enables usage of larger tiles, reducing the overhead. Scaling such a approach to a large cluster system with thousands of small computational results in unreasonable large disk usage. Based on the analysis in 5.2.4, utilization of our system for the $1 \times 10^9\,\mathsf{sample}$ input series in [2] increases the required disk-space and related overheads by a factor of 40. This justifies the effort for our target system. While adding properties like malleability and fault-tolerance to our approach based on MPI is possible, the required additional implementation efforts are high and could have been avoided by usage of different HPC frameworks.

In the retrospective, the *Charm++* programming model [77] appears as a potentially better choice: The code in this programming model is structured into *Chares*, modeling different tasks of the algorithm. Such Chares are dynamically assigned to the processing elements in a cluster for execution based on a message-driven control flow: Chares exchange data by messages to each other, and receival of such initiates execution of functionality.

This model provides all named benefits like fault-tolerance and malleability and also facilitates integration of accelerators like GPUs. Due to dynamic load-balancing of the runtime system, also a decomposition into irregular tiles is applicable, enabling heterogeneous computing. Tiles as in fig. 15a in the distance matrix are obvious candidates for creation of Chares. But a direct transfer of our presented parallelization scheme is not possible: to the best of our knowledge, mapping of the communication structure of sections 5.2.1 and 5.2.1 to Chares is impossible, as no equivalent to the multiple different MPI communicators used by each process exists. For this reason usage of reduction primitives to merge the results is not possible. It is undesirable for the model anyways, as a over-decomposition of different tiles is performed and their computation not finished simultaneously.

For a implementation based on *Charm++* [77], we propose to create two Chare arrays: one Chare array is used to keep the input time series and intermediate results in main memory, applying a partitioning like the I/O processes in our distributed implementation (sec. 5.2.1. A second Chare array constitutes a set of workers. With low overhead one master Chare can assign work partitions in terms of tile coordinates to idle Chares. Work tiles request required input sections from the I/O Chares and provide them with their intermediate results. Critical for efficiency of such a approach is the granularity of the work tiles: *Charm++* typically relies on a rather fine-grained decomposition to provide optimum load-balancing but highest efficiency of the matrix-profile computation requires sufficiently large work-packages (see sec. 5.2.4).

7.5 Summary and Outlook

We presented a solution to compute exact all-subsequence similarity-joins of time-series on a high performance computer cluster system with high efficiency. Enabled by the matrix-profile, it provides users with the ability to perform a variety of analysis on time series with

vastly increased detail or length. We examined the scaling behavior and provided a runtime model to predict efficient resource usage on a cluster system. With the availability of the concurrently presented SCAMP framework[2], we consider the scaling of the matrix profile approach a solved task.

As the solutions still become resource-intensive for large series, future work might consider to optimize the computations economically: while Zhu *et al.* [32] proposed a fast algorithm for approximate matrix profile computations, no guarantees on reliability are provided an implications on analysis not fully assessed. A different method to speed up time series analysis in the past for intractably large problems is sub-sampling which implies the loss of details and accuracy of the computation and for this reason requires expert knowledge about the data, high caution and can not be applied in general. In [78], a error bound for the euclidean distance is stated, if higher frequency components in a time series are dropped. Based on it, further research could investigate the implications of sub-sampling on analysis tasks like motif and discord discovery and potentially provide rules to reduce the amount of computations to the minimum required to solve the tasks. This is highly desirable, as the computation cost of $\Theta(n^2)$ potentiates any decrease of the input lengths n.

Sharing our work experiences with the MPI and Extra-P, we also hinted at problems and potential points of improvement, for example the option to investigate scaling behavior based on the minimum runtimes in Extra-P analysis.

Bibliography

[1] C. 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 for Time Series: A Unifying View that Includes Motifs, Discords and Shapelets,” in *2016 IEEE 16th International Conference on Data Mining (ICDM)*, Dec. 2016, pp. 1317–1322. DOI: 10.1109/ICDM.2016.0179.

[2] Z. Zimmerman, K. Kamgar, Y. Zhu, N. S. Senobari, B. Crites, G. Funning, and E. Keogh. (2018). Scaling Time Series Motif Discovery with GPUs: Breaking the Quintillion Pairwise Comparisons a Day Barrier. [Preprint], [Online]. Available at: https://www.cs.ucr.edu/%7Eeamonn/public/GPU%5C_Matrix%5C_profile%5C_VLDB%5C_30DraftOnly.pdf.

[3] C. Chatfield, *The analysis of time series*, 6. ed., ser. Texts in statistical science series. Boca Raton: Chapman & Hall/CRC, 2004, ISBN: 1584883170.

[4] T. Ferenti, “Biomedical applications of time series analysis,” in *2017 IEEE 30th Neumann Colloquium (NC)*, Nov. 2017, pp. 83–84. DOI: 10.1109/NC.2017.8263256.

[5] V. Kurbalija, C. von Bernstorff, H.-D. Burkhard, J. Nachtwei, M. Ivanovi, and L. Fodor, “Time-series mining in a psychological domain,” ACM Press, Sep. 16, 2012, p. 58, ISBN: 978-1-4503-1240-0. DOI: 10.1145/2371316.2371328.

[6] C. Cassisi, M. Aliotta, A. Cannata, P. Montalto, D. Patanè, A. Pulvirenti, and L. Spampinato, “Motif Discovery on Seismic Amplitude Time Series: The Case Study of Mt Etna 2011 Eruptive Activity,” *Pure and Applied Geophysics*, vol. 170, no. 4, pp. 529–545, Apr. 2013, ISSN: 1420-9136. DOI: 10.1007/s00024-012-0560-y.

[7] S. Moshiri and N. Cameron, “Neural network versus econometric models in forecasting inflation,” *Journal of Forecasting*, vol. 19, no. 3, pp. 201–217, Apr. 2000, ISSN: 0169-2070. [Online]. Available at: https://ssrn.com/abstract=114008.

[8] H. V. Haghi and S. M. M. Tafreshi, “An overview and verification of electricity price forecasting models,” in *2007 International Power Engineering Conference (IPEC 2007)*, Dec. 2007, pp. 724–729, ISBN: 978-981-05-9423-7.

[9] M. Hegland, W. Clarke, and M. Kahn, “Mining the MACHO dataset,” *Computer Physics Communications*, Conference on Computational Physics 2000: "New Challenges for the New Millenium", vol. 142, no. 1, pp. 22–28, Dec. 15, 2001, ISSN: 0010-4655. DOI: 10.1016/S0010-4655(01)00307-1. (visited on 10/07/2018).

[10] P. Chen, H. Yuan, and X. Shu, “Forecasting Crime Using the ARIMA model,” in *2008 Fifth International Conference on Fuzzy Systems and Knowledge Discovery*, vol. 5, Oct. 2008, pp. 627–630. DOI: 10.1109/FSKD.2008.222.

[11] J. Ilow, “Forecasting network traffic using FARIMA models with heavy tailed innovations,” in *2000 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings (Cat. No.00CH37100)*, vol. 6, Jun. 2000, 3814–3817 vol.6. DOI: 10.1109/ICASSP.2000.860234.

[12] A. Lerner, D. Shasha, Z. Wang, X. Zhao, and Y. Zhu, “Fast algorithms for time series with applications to finance, physics, music, biology, and other suspects,” in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data - SIGMOD '04*, Paris, France: ACM Press, Jun. 2004, pp. 965–968, ISBN: 978-1-58113-859-7. DOI: 10.1145/1007568.1007726.

[13] W.-K. Loh and J.-Y. Yun, “A parallel algorithm for robust fault detection in semiconductor manufacturing processes,” *Cluster Computing*, vol. 17, no. 3, pp. 643–651, Sep. 1, 2014, ISSN: 1386-7857, 1573-7543. DOI: 10.1007/s10586-014-0366-z. (visited on 08/09/2018).

[14] C. Kleist, “Time series data mining methods,” Humbold-Univeristät zu Berlin, Mar. 25, 2015. DOI: 10.18452/14237.

[15] J. F. Roddick and M. Spiliopoulou, “A survey of temporal knowledge discovery paradigms and methods,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 14, no. 4, pp. 750–767, Jul. 2002, ISSN: 1041-4347. DOI: 10.1109/TKDE.2002.1019212.

[16] E. Keogh, S. Lonardi, and B. Chiu, “Finding surprising patterns in a time series database in linear time and space,” in *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '02*, Edmonton, Alberta, Canada: ACM Press, 2002, p. 550, ISBN: 978-1-58113-567-1. DOI: 10.1145/775047.775128.

[17] S. Torkamani and V. Lohweg, “Survey on time series motif discovery: Time series motif discovery,” *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 7, Feb. 13, 2017. DOI: 10.1002/widm.1199.

[18] Z. Liu, J. X. Yu, X. Lin, H. Lu, and W. Wang, “Locating Motifs in Time-Series Data,” in *Advances in Knowledge Discovery and Data Mining*, ser. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, May 18, 2005, pp. 343–353, ISBN: 978-3-540-26076-9. DOI: 10.1007/11430919_41.

[19] A. Mueen, “Time series motif discovery: Dimensions and applications: Time series motif discovery,” *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 4, no. 2, pp. 152–159, Mar. 2014, ISSN: 19424787. DOI: 10.1002/widm.1119.

[20] H. A. Dau and E. Keogh, “Matrix Profile V: A Generic Technique to Incorporate Domain Knowledge into Motif Discovery,” in *Proceedings of the 23rd International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, ACM Press, 2017, pp. 125–134, ISBN: 978-1-4503-4887-4. DOI: 10.1145/3097983.3097993.

[21] C.-C. M. Yeh, Y. Zhu, L. Ulanova, N. Begum, Y. Ding, H. A. Dau, Z. Zimmerman, D. F. Silva, A. Mueen, and E. Keogh, "Time series joins, motifs, discords and : A unifying view that exploits the matrix profile," *Data Mining and Knowledge Discovery*, vol. 32, no. 1, pp. 83–123, Jan. 2018, ISSN: 1573-756X. DOI: 10.1007/s10618-017-0519-9.

[22] C. C. M. Yeh, N. Kavantzas, and E. Keogh, "Matrix profile VI: Meaningful multidimensional motif discovery," in *2017 IEEE International Conference on Data Mining (ICDM)*, Nov. 2017, pp. 565–574. DOI: 10.1109/ICDM.2017.66.

[23] E. Keogh, J. Lin, S.-H. Lee, and H. V. Herle, "Finding the most unusual time series subsequence: Algorithms and applications," *Knowledge and Information Systems*, vol. 11, pp. 1–27, Dec. 4, 2006, ISSN: 0219-1377, 0219-3116. DOI: 10.1007/s10115-006-0034-6.

[24] C. Varun, C. Deepthi, and K. Vipin, "Detecting anomalies in a time series database," Univeristy of Minnesota, Feb. 5, 2009. [Online]. Available at: https://www.cs.umn.edu/research/technical_reports/view/09-004 (visited on 08/11/2018).

[25] Y. Zhu, M. Imamura, D. Nikovski, and E. Keogh, "Matrix profile VII: Time series chains: A new primitive for time series data mining (best student paper award)," in *2017 IEEE International Conference on Data Mining (ICDM)*, New Orleans, LA: IEEE, Nov. 2017, pp. 695–704, ISBN: 978-1-5386-3835-4. DOI: 10.1109/ICDM.2017.79.

[26] Y. Zhu, M. Imamura, D. Nikovski, and E. Keogh, "Introducing time series chains: A new primitive for time series data mining," *Knowledge and Information Systems*, Jun. 2018, ISSN: 0219-3116. DOI: 10.1007/s10115-018-1224-8.

[27] C. 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)*, Dec. 2016, pp. 579–588. DOI: 10.1109/ICDM.2016.0069.

[28] E. Keogh and J. Lin, "Clustering of time-series subsequences is meaningless: implications for previous and future research," *Knowledge and Information Systems*, vol. 8, pp. 154–177, Aug. 2005, ISSN: 0219-1377. DOI: 10.1007/s10115-004-0172-7.

[29] C.-C. M. Yeh, N. Kavantzas, and E. Keogh, "Matrix profile IV: Using weakly labeled time series to predict outcomes," *Proceedings of the VLDB Endowment*, vol. 10, pp. 1802–1812, Aug. 1, 2017, ISSN: 21508097. DOI: 10.14778/3137765.3137784.

[30] A. Mueen, Y. Zhu, C.-C. M. Yeh, K. Kamgar, K. Vishwanathan, C. Gupta, and E. Keogh. (Aug. 2017). The fastest similarity search algorithm for time series subsequences under euclidean distance, [Online]. Available at: http://www.cs.unm.edu/~mueen/FastestSimilaritySearch.html (visited on 08/21/2018).

[31] S. Zilberstein and S. Russell, "Approximate Reasoning Using Anytime Algorithms," in *Imprecise and Approximate Computation*, S. Natarajan, Ed., vol. 318, Boston, MA: Springer US, 1995, pp. 43–62, ISBN: 978-0-7923-9579-9. DOI: 10.1007/978-0-585-26870-5_4.

[32] 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)*, Nov. 2018, pp. 837–846. DOI: 10.1109/ICDM.2018.00099.

[33] T. Huang, Y. Zhu, Y. Mao, X. Li, M. Liu, Y. Wu, Y. Ha, and G. Dobbie, "Parallel Discord Discovery," in *Advances in Knowledge Discovery and Data Mining*, ser. Lecture Notes in Computer Science, Springer, Cham, Apr. 19, 2016, pp. 233–244, ISBN: 978-3-319-31749-6. DOI: 10.1007/978-3-319-31750-2_19.

[34] E. Keogh, J. Lin, and A. Fu, "HOT SAX: Efficiently finding the most unusual time series subsequence," in *Fifth IEEE International Conference on Data Mining (ICDM'05)*, Nov. 2005, ISBN: 0-7695-2278-5. DOI: 10.1109/ICDM.2005.79.

[35] A. Berard and G. 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*, Chicago, Illinois: ACM Press, 2013, pp. 15–22, ISBN: 978-1-4503-2324-6. DOI: 10.1145/2501221.2501224.

[36] A. Movchan and M. L. Zymbler, "Time Series Subsequence Similarity Search Under Dynamic Time Warping Distance on the Intel Many-core Accelerators," in *Similarity Search and Applications*, R. Amato Giuseppe and Connor, F. Falchi, and C. Gennaro, Eds., Cham: Springer International Publishing, 2015, pp. 295–306, ISBN: 978-3-319-25087-8. DOI: 10.1007/978-3-319-25087-8_28.

[37] Y. Zhu, Z. Zimmerman, N. S. Senobari, C. M. Yeh, G. Funning, A. Mueen, P. Brisk, and E. Keogh, "Matrix profile II: Exploiting a novel algorithm and GPUs to break the one hundred million barrier for time series motifs and joins," in *2016 IEEE 16th International Conference on Data Mining (ICDM)*, Dec. 2016, pp. 739–748. DOI: 10.1109/ICDM.2016.0085.

[38] J. Lin, E. Keogh, S. Lonardi, and P. Patel, "Finding motifs in time series," in *2nd Workshop on Temporal Data Mining*, Edmonton, Alberta, Canada, 2002. [Online]. Available at: https://www.cs.ucr.edu/~stelo/papers/motif_KDD.pdf.

[39] A. Mueen and E. Keogh. Time Series Data Mining Using the Matrix Profile: A Unifiying View of Motif Discovery, Anomaly Detection, Segmentation, Classification, Clustering and Similarity Joins, [Online]. Available at: http://www.cs.ucr.edu/%7Eeamonn/Matrix_Profile_Tutorial_Part2.pdf (visited on 08/18/2018).

[40] X. Wang, A. Mueen, H. Ding, G. Trajcevski, P. Scheuermann, and E. Keogh, "Experimental comparison of representation methods and distance measures for time series data," *Data Mining and Knowledge Discovery*, vol. 26, no. 2, pp. 275–309, Mar. 1, 2013, ISSN: 1384-5810, 1573-756X. DOI: 10.1007/s10618-012-0250-5.

[41] E. Keogh and S. Kasetty, "On the Need for Time Series Data Mining Benchmarks: A Survey and Empirical Demonstration," *Data Mining and Knowledge Discovery*, vol. 7,

no. 4, pp. 349–371, Oct. 1, 2003, ISSN: 1384-5810, 1573-756X. DOI: 10.1023/A:1024988512476.

[42] T. Rakthanmanon, B. Campana, A. Mueen, G. Batista, B. Westover, Q. Zhu, J. Zakaria, and E. Keogh, “Searching and mining trillions of time series subsequences under dynamic time warping,” in *Proceedings of the 18th international conference on Knowledge discovery and data mining (SIGKDD)*, ACM Press, 2012, pp. 262–270, ISBN: 978-1-4503-1462-6. DOI: 10.1145/2339530.2339576.

[43] K. Hwang and Z. Xu, *Scalable parallel computing: technology, architecture, programming*. Boston: WCB/McGraw-Hill, 1998, 802 pp., ISBN: 978-0-07-031798-7.

[44] T. Rauber and G. Rünger, *Parallel programming: for multicore and cluster systems*, second edition. Berlin Heidelberg: Springer, 2013, ISBN: 978-3-642-37801-0.

[45] R. Agrawal, C. Faloutsos, and A. Swami, “Efficient similarity search in sequence databases,” in *Foundations of Data Organization and Algorithms*, D. B. Lomet, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 69–84, ISBN: 978-3-540-48047-1.

[46] M. J. Quinn, *Parallel programming in C with MPI and OpenMP*, Internat. ed. 2003, [Nachdr.] Boston, Mass. [u.a]: McGraw-Hill, 2008, ISBN: 978-0-07-123265-4.

[47] J. McCalpin, “Memory bandwidth and machine balance in high performance computers,” *IEEE Technical Committee on Computer Architecture Newsletter*, pp. 19–25, Dec. 1, 1995.

[48] J. Weidendorfer, M. Kowarschik, and C. Trinitis, “A Tool Suite for Simulation Based Analysis of Memory Access Behavior,” in *Computational Science - ICCS 2004*, vol. 3038, Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 440–447, ISBN: 978-3-540-22116-6. DOI: 10.1007/978-3-540-24688-6_58.

[49] D. Terpstra, H. Jagode, H. You, and J. Dongarra, “Collecting performance data with PAPI-c,” in *Tools for High Performance Computing 2009*, M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 157–173, ISBN: 978-3-642-11260-7. DOI: 10.1007/978-3-642-11261-4_11.

[50] D. Marques, H. Duarte, A. Ilic, L. Sousa, R. Belenov, P. Thierry, and Z. A. Matveev, “Performance Analysis with Cache-Aware Roofline Model in Intel Advisor,” in *2017 International Conference on High Performance Computing Simulation (HPCS)*, Genoa, Italy, Jul. 2017, pp. 898–907. DOI: 10.1109/HPCS.2017.150.

[51] A. Ilic, F. Pratas, and L. Sousa, “Cache-aware Roofline model: Upgrading the loft,” *IEEE Computer Architecture Letters*, vol. 13, no. 1, pp. 21–24, Jan. 2014, ISSN: 1556-6056. DOI: 10.1109/L-CA.2013.6.

[52] G. Ofenbeck, R. Steinmann, V. Caparros, D. G. Spampinato, and M. Puschel, “Applying the roofline model,” in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, Mar. 2014, pp. 76–85, ISBN: 978-1-4799-3606-9. DOI: 10.1109/ISPASS.2014.6844463.

[53] T. Koskela, Z. Matveev, C. Yang, A. Adedoyin, R. Belenov, P. Thierry, Z. Zhao, R. Gayatri, H. Shan, L. Oliker, J. Deslippe, R. Green, and S. Williams, “A novel multi-level integrated roofline model approach for performance characterization,” in *High Performance Computing*, R. Yokota, M. Weiland, D. Keyes, and C. Trinitis, Eds., vol. 10876, Cham: Springer International Publishing, 2018, pp. 226–245, ISBN: 978-3-319-92040-5. DOI: 10.1007/978-3-319-92040-5_12.

[54] U. R. Drepper, *What every programmer should know about memory*, Nov. 21, 2007. [Online]. Available at: https://akkadia.org/drepper/cpumemory.pdf.

[55] Linux Kernel Organization, Inc. Perf: Linux profiling with performance counters, [Online]. Available at: https://perf.wiki.kernel.org/index.php/Main_Page (visited on 01/09/2019).

[56] J. Treibig, G. Hager, G. Wellein, and M. Meier, “LIKWID: Lightweight performance tools,” in *Proceedings of the 2011 companion on High Performance Computing Networking, Storage and Analysis Companion - SC ’11 Companion*, Seattle, Washington, USA: ACM Press, 2011, p. 29, ISBN: 978-1-4503-1030-7. DOI: 10.1145/2148600.2148616.

[57] A. H. Karp and H. P. Flatt, “Measuring parallel processor performance,” *Communications of the ACM*, vol. 33, pp. 539–543, May 1, 1990, ISSN: 00010782. DOI: 10.1145/78607.78614.

[58] N. Perlin, J. P. Zysman, and B. P. Kirtman, “Practical scalability assesment for parallel scientific numerical applications,” *CoRR*, vol. abs/1611.01598, 2016. arXiv: 1611.01598.

[59] A. Calotoiu, T. Hoefler, M. Poke, and F. Wolf, “Using automated performance modeling to find scalability bugs in complex codes,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC ’13*, Denver, Colorado: ACM Press, 2013, pp. 1–12, ISBN: 978-1-4503-2378-9. DOI: 10.1145/2503210.2503277. (visited on 10/22/2018).

[60] A. Knüpfer, C. Rössel, D. a. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf, “Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir,” in *Tools for High Performance Computing 2011*, H. Brunst, M. S. Müller, W. E. Nagel, and M. M. Resch, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 79–91, ISBN: 978-3-642-31475-9 978-3-642-31476-6. DOI: 10.1007/978-3-642-31476-6_7.

[61] E. A. Carmona and M. D. Rice, “Modeling the serial and parallel fractions of a parallel algorithm,” *Journal of Parallel and Distributed Computing*, vol. 13, no. 3, pp. 286–298, Nov. 1, 1991, ISSN: 0743-7315. DOI: 10.1016/0743-7315(91)90076-L.

[62] A. Grama, A. Gupta, and V. Kumar, “Isoefficiency function: A scalability metric for parallel algorithms and architectures,” *IEEE Parallel and Distributed Technology, Special Issue on Parallel and Distributed Systems: From Theory to Practice*, vol. 1, pp. 12–21, 1993.

[63] Y. Zhu, C.-C. M. Yeh, Z. Zimmerman, K. Kamgar, and E. Keogh. SCRIMP++: Motif Discovery at Interactive Speeds, [Online]. Available at: https://sites.google.com/site/scrimpplusplus/ (visited on 09/11/2018).

[64] O. A. R. Board, *OpenMP application program interface version 4.0*, Jul. 2013. [Online]. Available at: https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf (visited on 12/12/2018).

[65] Intel Corporation. Intel intrinsics guide, [Online]. Available at: https://software.intel.com/sites/landingpage/IntrinsicsGuide/ (visited on 12/12/2018).

[66] Intel Corporation, *Intel Advanced Vector Extensions Programming Reference*, no. 319433-011, Jun. 2011. [Online]. Available at: https://software.intel.com/sites/default/files/m/f/7/c/36945.

[67] MPI Forum, *MPI: A Message-Passing Interface Standard, Version 3.1*, first edition. Stuttgart: High Performance Computing Center, Jun. 4, 2015. [Online]. Available at: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf (visited on 06/21/2018).

[68] R. Latham, R. Ross, and R. Thakur, “The Impact of File Systems on MPI-IO Scalability,” in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, D. Kranzlmüller, P. Kacsuk, and J. Dongarra, Eds., red. by D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, and G. Weikum, vol. 3241, Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 87–96, ISBN: 978-3-540-23163-9. DOI: 10.1007/978-3-540-30218-6_18.

[69] LRZ: SuperMUC petascale system, [Online]. Available at: https://www.lrz.de/services/compute/supermuc/systemdescription/ (visited on 09/22/2018).

[70] B. Huang, M. Bauer, and M. Katchabaw, “Hpcbench - a linux-based network benchmark for high performance networks,” in *19th International Symposium on High Performance Computing Systems and Applications (HPCS’05)*, May 2005, pp. 65–71, ISBN: 0-7695-2343-9. DOI: 10.1109/HPCS.2005.32.

[71] J. Treibig, G. Hager, and G. Wellein, “LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments,” in *2010 39th International Conference on Parallel Processing Workshops*, Sep. 2010, pp. 207–216. DOI: 10.1109/ICPPW.2010.38.

[72] D. Molka, R. Schöne, D. Hackenberg, and W. E. Nagel, “Detecting memory-boundedness with hardware performance counters,” in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering - ICPE ’17*,

L'Aquila, Italy: ACM Press, 2017, pp. 27–38, ISBN: 978-1-4503-4404-3. DOI: 10.1145/3030207.3030223.

[73] E. Jones, T. Oliphant, P. Peterson, *et al.*, *SciPy: Open source scientific tools for Python*. 2001. [Online]. Available at: http://www.scipy.org/ (visited on 01/11/2019).

[74] A. D. Lemire. (Jan. 2018). Microbenchmarking calls for idealized conditions, Daniel Lemire's blog, [Online]. Available at: https://lemire.me/blog/2018/01/16/microbenchmarking-calls-for-idealized-conditions/ (visited on 12/21/2018).

[75] W. D. Gropp, E. Lusk, and A. Skjellum, *Using Advanced MPI: Modern Features of the Message-Passing Interface*. Cambridge, Mass.; London: MIT Press, 2015, OCLC: 925227020, ISBN: 978-0-262-32663-6.

[76] W. Zhang, M. Hao, and M. Snir, "Predicting HPC parallel program performance based on LLVM compiler," *Cluster Computing*, vol. 20, pp. 1179–1192, Jun. 2017, ISSN: 1386-7857. DOI: 10.1007/s10586-016-0707-1.

[77] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, and L. Kale, "Parallel Programming with Migratable Objects: Charm++ in Practice," in *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2014, pp. 647–658. DOI: 10.1109/SC.2014.58.

[78] A. Mueen, S. Nath, and J. Liu, "Fast approximate correlation for massive time-series data," in *Proceedings of the 2010 international conference on Management of data - SIGMOD '10*, Indianapolis, Indiana, USA: ACM Press, 2010, p. 171, ISBN: 978-1-4503-0032-2. DOI: 10.1145/1807167.1807188.

List of Figures

1 Example of a motif in a time series [17] 10
2 Subsequence clustering and time series chain examples, from [26], [27] 10
3 Time series A of length n and its subsequences of length m 16
4 Definition of the matrix profile and index 18
5 Matrix Profile in case of a self-similarity search 19
6 Data layout 21
7 Scrimp iteration scheme 23
8 Example of a roofline diagram [50] 25
9 Inputs and outputs of the algorithmic kernel 32
10 Vertical blocking scheme 35
11 Access pattern of vertical blocking 38
12 Partitioning of work among processes 44
13 Binary time series file format 47
14 Binary matrix profile file format 49
15 Partitioning for distributed parallelization 52
16 Padding of the time series input 55
17 Evaluation of upper tiles 57
18 Input partitioning and communciation 58
19 Merging partial result slices 59
20 Special cases of result merging 60
21 Output processes and partitioning 62
22 SCAMP processing scheme [2] 66
23 Roofline diagram comparing different kernels 69
24 Comparison of sequential kernels 70
25 Variation of the block length 74
26 Scaling behavior of trival parallalization 77
27 Overheads of the trival parallalization 79
28 Performance models in strong and weak scaling 85
29 Overheads in strong scaling with the distributed parallelization 88
30 Scaling of I/O functions 90

31 Overheads of the distributed parallalization in weak scaling 92

32 Scaling behaviour of MPI_File_Open 94

33 Comparision of implementations in strong scaling 95

34 Throughput comparision, including SCAMP 97

35 Network bandwidth measurement using Hpcbench 121

36 Accumulate idle-times after evaluations in strongscaling 131

List of Tables

1 Number of memory accesses of the kernel 34
2 Arithmetic intensities of the kernel 34
3 Memory accesses in a iteration over a vertical block 39
4 Theoretical scaling analysis of trivial parallelization 50
5 Theoretical scaling analysis of distributed implementation 64
6 Roofline data points of kernels: 69
7 Performance counter measurements of vectorized kernels 73
8 Model fit of trivial parallelization behavior 78
9 Scaling behavior of selected callpaths of the trivial parallelization 81
10 Goodness of performance model fit for distributed parallelization: 83
11 Large-scale runtime predictions 84
12 Strong scaling behavior of selected functions of distributed parallelization 91
13 Model-based comparison of different implementations 99
14 System setup of SuperMUC 118
15 Processing hardware of SuperMUC 119
16 Software Environment at SuperMUC 120
17 Local Software environemnt for result analysis 120
18 Comparison of kernel throughputs: 121
19 Impact of the block length on kernel throughput 122
20 Block length variation with large input 123
21 Strong scaling with trivial parallelization 123
22 Modeling experiment with distributed parallelization 124
23 Detailed timings of large-scale runs: 125
24 Weak scaling behavior of selected sections as reported by Extra-P 126
25 Fitting quality for comparison of different implementations 127
26 Comparison of implementations based on strong scaling 127

8. Appendix

8.1 Experimental Setup

If not stated differently, all experiments in our work are performed on the SuperMUC Phase II cluster system [69].

SuperMUC Phase II

The system is hierarchically structured into islands and nodes. The finest granularity of job allocations are single nodes. Hence programs running on less than the number of cores on a node get the full resources available to one node. Table 15 and 14 list hardware information of the system [69]. Note that at the time of our experiments the systems operating frequency was reduced by the system administrators[1]. The stated peak performances and memory bandwidths are the mean of three values extracted from Intel Advisor roofline diagrams. Those values are obtained by micro-benchmarks which are run in advance to the application analysis [51] and for this reason represent practically achievable hardware limits.

Table 14 System setup of the SuperMUC phase II system: System structure, network and storage information

Processors per node	2
Nodes per island	512
Number of islands	6
Interconnect	Infiniband FDR14
Intra-Island topology	non-blocking Tree
Inter-Island topology	Pruned tree
Bisection interconnect bandwidth	5.1 TByte/s
Shared aggregate bandwidth to/from parallel storage	250 GByte/s

[1] Information might be available on https://www.lrz.de/aktuell/ali00687.html

Table 15 Processing hardware information of SuperMUC Phase II system

System	Lenovo NeXtScale nx360M5 WCT
Processors Processor family	Intel(R) Xeon(R) CPU E5-2697 v3 Intel Xeon Haswell EN/EP/EX processor
Nominal frequency Operating frequency	2.60 GHz 1.8 GHz
Peak performance 1 core DP FMA DP Vector Add DP Scalar Add	 28.7 GFLOPS 7.18 GFLOPS 1.80 GFLOPS
Cores per CPU socket	14
Sockets per node	2
NUMA domains per Node	4
Cache levels	3
core private cache: Level 1 Level 2	 32 kB 256 kB
NUMA domain shared Cache Level 3	 18 MB
Main memory per NUMA Domain node total bandwidth (28 threads)	 16 GB 64 GB 128.3 GB/s

Table 16 Software and build environment used at the SuperMUC phase II system

Operating system Kernel version:	Suse Linux Enterprise Server 3.0.101-108.48-default
C Compiler	Intel ICC 18.0.2 20180210
MPI	Intel MPI for Linux* OS, Version 2018 Update 2 Build 20180125
LIKWID	Version 4.3.0
PAPI	Version 5.6.0.0
Boost	Version 1.65.1
FFTW	Version 3.3.3
GCC (STL)	Version 7.3.0
CMake	version 3.8.2
Intel(R) Advisor	Advisor 2018 Update 2
Score-P	Version 3.0
LIKWID	Version 4.3.0

Evaluation Software

Table 17 Local Software environemnt for result analysis

Extra-P:	Version 3.0
Python:	Anaconda Python 3.6.0
SciPy:	Version 1.0.0

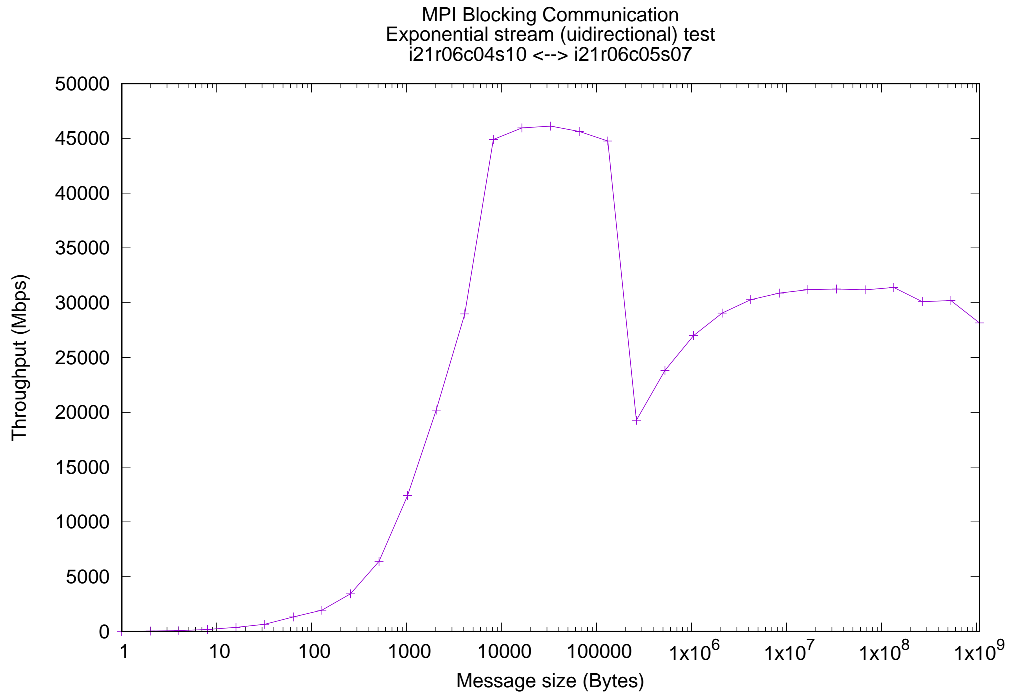


Figure 35 Network bandwidth measurement using Hpcbench [70]: network bandwidth of blocking (Intel) MPI network communication between two nodes measured for varying message sizes. Reported round-trip-time in separate experiment using 64-byte messages, repeated 10 times: min/avg/max = 5.505/5.700/5.759 µs

8.2 Numerical Results

Table 18 Comparison of kernel throughputs: listed are statistics of the experiment of figure 24. The original SCRIMP kernel of Zhu *et al.* [32] and the three kernels discussed in chapter 4.1 are measured, the experiment is described in section 6.1.1. The window length was set to 1×10^3 samples for matrix profile computation with two different length input series. Statistics are computed from 5 repeated experiments.
Runtime statistics over all experiments (mean, max, min) in s: (1.46×10^3; 5.49×10^3; 6.63)

	input length 100×10^3 **throughput** / entries/s		**input length** 1×10^6 **throughput** / entries/s	
kernel	**mean**	**std.**	**mean**	**std**
original	216×10^6	354×10^3	179×10^6	452×10^3
arithm	436×10^6	17.3×10^6	282×10^6	2.94×10^6
intrins	467×10^6	591×10^3	490×10^6	89.5×10^3
autovec	1.22×10^9	1.73×10^6	1.57×10^9	1.74×10^6

Table 19 Impact of the block length on kernel throughput: Data of the experiment plotted in figures 25a and 25b. For each of the listed block lengths, measurement of the kernel throughput was repeated two times with each of the subsequence search lengths $w \in \{50, 100, 1000, 4096\}$. Used input data was a random walk time series with 100×10^3 samples.
Runtime statistics over all experiments for the autovec kernel, (mean, max, min) in s: (8.968189; 19.858263; 6.665488)
Runtime statistics over all experiments for the intrinsics kernel, (mean, max, min) in s: (20.7; 28.0; 16.9)

	compute throughput / matrix entries/s			
	autovec kernel		**intrinsics kernel**	
block-length	**mean**	**std.**	**mean**	**std**
10	520×10^6	1.38×10^6	366×10^6	6.13×10^6
50	1.13×10^9	10.2×10^6	470×10^6	18.2×10^6
100	1.35×10^9	13.8×10^6	515×10^6	48.2×10^6
300	1.55×10^9	20.9×10^6	519×10^6	50.0×10^6
500	1.60×10^9	21.1×10^6	521×10^6	50.6×10^6
600	1.55×10^9	26.7×10^6	519×10^6	51.1×10^6
700	1.50×10^9	42.1×10^6	517×10^6	51.8×10^6
800	1.45×10^9	50.7×10^6	517×10^6	52.3×10^6

Table 20 Block length variation with large input: additional data of the experiment plotted in figures 25c. For each block-length the measurement of the kernel throughput is performed once with each of the search window lengths $w \in \{1000, 4096\}$. Used input data is a random walk time series with 1×10^{6} samples. As only two measurements are been performed, the reported standard deviation is degenerated to the absolute difference Δ of the throughputs.
Runtime statistics over all experiments for the intrinsics kernel, (mean, max, min) in s: (863; 1.96×10^{3}; 624)

	compute throughput / matrix entries/s	
block-length	**mean**	Δ
10	513×10^{6}	508×10^{3}
50	1.12×10^{9}	76.1×10^{3}
100	1.36×10^{9}	581×10^{3}
300	1.56×10^{9}	578×10^{3}
500	1.62×10^{9}	2.33×10^{6}
600	1.54×10^{9}	4.64×10^{6}
700	1.47×10^{9}	2.32×10^{6}
800	1.41×10^{9}	4.89×10^{6}

Table 21 Strong scaling with trivial parallelization: Strong scaling with random walk input time series of length 801249 using a subsequence search window length of 1000 samples. With each number of processes 7 measurements have been taken. See figure 26b

p	**mean** T_{total} / s	**median** T_{total} / s	**std.** T_{total} **/ s**	**mean** $T_{\mathrm{work,total}}$ / s	**mean** $T_{\mathrm{comp,total}}$ / s
1	1.331×10^{3}	1.331×10^{3}	1.823×10^{-1}	1.330×10^{3}	1.330×10^{3}
36	1.401×10^{3}	1.401×10^{3}	1.227×10^{1}	1.374×10^{3}	1.317×10^{3}
81	1.558×10^{3}	1.557×10^{3}	3.094×10^{1}	1.473×10^{3}	1.305×10^{3}
144	1.617×10^{3}	1.608×10^{3}	6.828×10^{1}	1.445×10^{3}	1.350×10^{3}
289	1.949×10^{3}	1.960×10^{3}	5.758×10^{1}	1.636×10^{3}	1.313×10^{3}
529	2.531×10^{3}	2.522×10^{3}	6.371×10^{1}	1.907×10^{3}	1.361×10^{3}
1089	5.015×10^{3}	5.049×10^{3}	3.802×10^{2}	2.790×10^{3}	1.478×10^{3}

Table 22 Modeling experiment with distributed parallelization: Listed are the parallel runtimes of the distributed parallelization with varying process number p and different input sizes. Similarity search was performed with a window length of 1000 samples. The problem size is stated in terms of the number of distance matrix entries relative to the smallest problem. The smallest problem uses inputs of length 89916, yielding a distance matrix with 6.400×10^{11} entries. The relative problem size refers to that smallest one (see eqn. 6.2). Statistics for each measurement point are computed from 6 repeated measurements.

p	rel. prob. size	mean runtime / s	median runt. / s	std. runt. / s
81	1	1.591×10^{1}	1.594×10^{1}	8.246×10^{-2}
81	1.8	2.805×10^{1}	2.799×10^{1}	1.422×10^{-1}
81	3.6	5.574×10^{1}	5.569×10^{1}	1.117×10^{-1}
81	6.5	1.017×10^{2}	1.017×10^{2}	1.061×10^{-1}
81	1.3×10^{1}	2.090×10^{2}	2.089×10^{2}	2.317×10^{-1}
144	1	9.269	9.265	7.043×10^{-2}
144	1.8	1.668×10^{1}	1.615×10^{1}	9.373×10^{-1}
144	3.6	3.180×10^{1}	3.179×10^{1}	8.166×10^{-2}
144	6.5	5.751×10^{1}	5.746×10^{1}	9.475×10^{-2}
144	1.3×10^{1}	1.181×10^{2}	1.181×10^{2}	2.143×10^{-1}
289	1	4.766	4.724	8.260×10^{-2}
289	1.8	8.466	8.209	5.579×10^{-1}
289	3.6	1.607×10^{1}	1.614×10^{1}	1.175×10^{-1}
289	6.5	2.958×10^{1}	2.935×10^{1}	6.405×10^{-1}
289	1.3×10^{1}	5.976×10^{1}	5.948×10^{1}	6.166×10^{-1}
529	1	3.063	2.889	4.019×10^{-1}
529	1.8	4.783	4.766	1.075×10^{-1}
529	3.6	9.329	9.326	1.297×10^{-1}
529	6.5	1.638×10^{1}	1.624×10^{1}	3.237×10^{-1}

p	rel. prob. size	mean runtime / s	median runt. / s	std. runt. / s
529	1.3×10^{1}	3.284×10^{1}	3.275×10^{1}	2.380×10^{-1}
1089	1	1.902	1.925	1.123×10^{-1}
1089	1.8	2.836	2.780	2.278×10^{-1}
1089	3.6	5.006	4.999	1.507×10^{-1}
1089	6.5	8.648	8.598	2.402×10^{-1}
1089	1.3×10^{1}	1.697×10^{1}	1.693×10^{1}	3.762×10^{-1}

Table 23 Detailed timings of large-scale runs: shown are timings measured in our runtime prediction tests with the distributed parallelization described in section 6.3.1 as well as available data from the SCAMP publication [2]. The listed times are accumulated over all processes and stem from a single experiment each.

implementation:	distrib. par	distrib. par	SCAMP
input length	6.40×10^{6}	100×10^{6}	1.00×10^{9}
Cost /h	28.4	5.71×10^{3}	376
computation time /h	27.6	5.71×10^{3}	375
overhead time /h	833×10^{-3}	4.98	1.00
par. efficiency /%	97.1	99.9	99.7
eval time /h	27.5	5.71×10^{3}	
precomp time /h	20.4×10^{-3}	276×10^{-3}	
comm time /h	525×10^{-3}	4.41	
IO time /h	284×10^{-3}	453×10^{-3}	
others /h	24.2×10^{-3}	126×10^{-3}	

Table 24 Weak scaling behavior of selected sections as reported by Extra-P for the distributed parallelization. Each measurement was performed 7 times. Weak scaling was started with a input length of 89.9×10^3 samples for 1 process. Scaling increased the relative problem size (see sec. 6.3) equally to the processor number up to 1.09×10^3 processes. Average parallel runtimes ranged from 15.7 s to 17.3 s. The subsequence window length was set to $w = 1.00\times10^3$ samples. The sections *accu_writing* and *accu_idle_eval* are manually recorded timings of program sections, which we parsed in a textfile to Extra-P (see sec. 6.3). Further we list a few selected interesting MPI function calls from the set of recorded Score-P profiles. Using the CLI *extrap-modeler* utility we examined fitting to means and medians separately, which show highly different results due to generally high variances in the data set.

	accu. time model / s	$R^2/\%$	**SMAPE** / $\%$
	mean fitting		
accu_writing	$56.5 + 9.82 \cdot p^2 \log_2^2(p)$	99.6	28.3
accu_idle_eval	$2.231 + 3.78 \cdot p^{1.5}$	1	3.75
MPI_File_open (input)	$0.029 + 8.81\times10^{-12} \cdot p^3 \log_2^2(p)$	99.8	47.8
MPI_File_read_all	$0.123 + 2.27\times10^{-3} \cdot p$	98.3	39.2
MPI_File_open (output)	$0.39 + 3.33\times10^{-9} \cdot p^3$	97.9	48.9
MPI_File_write_all	$0.58 + 1.30\times10^{-3} \cdot p \log_2 p$	96.8	53.6
MPI_Barrier	$47.9 + 1.25\times10^{-3} \cdot p^2$	99.6	52.6
	median fitting		
accu_writing	$56.2 + 6.10(p^2)$	98.1	35.3
accu_idle_eval	$6.38 + 1.31\times10^{-6} \cdot p^2 \log_2(p)$	99.8	16.7
MPI_File_open (input)	$0.028 + 285\times10^{-9} \cdot p^2$	98.5	41.3
MPI_File_read_all	$0.14 + 2.16\times10^{-3} \cdot p$	97.7	39.5
MPI_File_open (output)	$0.21 + 3.42\times10^{-9} p^3$	99.4	50.6
MPI_File_write_all	$0.63 + 9.26\times10^{-3} \cdot p$	95.2	50.7
MPI_Barrier	$-8.5 + 10\times10^{-3} \cdot p \log_2^2(p)$	99.7	37.8

Table 25 Fitting quality for comparison of different implementations: additional error metrics for the fitted models in table 13, which we omitted due to the space restriction

implementation	RMSE /s	$R^2/1$	SMAPE /%
trivial par.	170×10^{-3}	1	35×10^{-3}
distrib. par	62×10^{-3}	1	14×10^{-3}
SCAMP CPU	2.1	1000×10^{-3}	650×10^{-3}
SCAMP GPU	90×10^{-3}	1000×10^{-3}	370×10^{-3}

Table 26 Comparison of implementations based on strong scaling: strong scaling with random walk input time series of length 801249 using a subsequence search window length of 1000 samples. Mean values over 7 experiments, as plotted in figure 33. Listed are mean values of the parallel cost for the distributed parallelization without (*distrib. raw*) and with Score-P instrumentation and synchronization barriers (*distr. Score-P*). As they are obtained from the modeling experiment (see tab. 22) and a additional seqeutial run, no data for $p = 36$ is available. Further timings for the trivial parallelization with instrumentation are shown. Relative runtime overhead (*rel. overhead*) of the insturmentation is computed as $(t_{\mathrm{ScoreP}}(p) - t_{\mathrm{raw}}(p))/t_{\mathrm{raw}}(p)$ for the distributed parallelization. Relative speedup of the distributed parallelization against the trivial one is computed by $t_{\mathrm{triv}}(p)/t_{\mathrm{distrib}}(p)$ for each p

p	distrib. raw $\overline{T_{\mathrm{total}}}$ / s	distrib. Score-P $\overline{T_{\mathrm{total}}}$/s	triv Score-P $\overline{T_{\mathrm{total}}}$/s	rel. overhead /%	rel. speedup /1
1	1.254×10^{3}	1.311×10^{3}	1.331×10^{3}	4.5	1.0
36		1.348×10^{3}	1.401×10^{3}		1.0
81	1.289×10^{3}	1.365×10^{3}	1.558×10^{3}	5.9	1.1
144	1.335×10^{3}	1.423×10^{3}	1.617×10^{3}	6.6	1.1
289	1.377×10^{3}	1.494×10^{3}	1.949×10^{3}	8.5	1.3
529	1.620×10^{3}	1.714×10^{3}	2.531×10^{3}	5.8	1.5
1089	2.071×10^{3}	2.281×10^{3}	4.977×10^{3}	10	2.2

8.3 Textual Listings

Listing 8.1: Score-P filter file for application profiling

```
1  SCOREP_FILE_NAMES_BEGIN
2    EXCLUDE *
3    INCLUDE *Scrimp*.cpp
4    INCLUDE *bintsfile.cpp
5    INCLUDE *binproffile.cpp
6    INCLUDE *biniobase.cpp
7  SCOREP_FILE_NAMES_END
8
9  SCOREP_REGION_NAMES_BEGIN
10   EXCLUDE std::*
11   EXCLUDE matrix_profile::ScrimpDistribPar::eval_diag_block_triangle
12 SCOREP_REGION_NAMES_END
```

Listing 8.2: Extra-P analysis of program sections in strong scaling. The analyzed times are those presented in figure 29. The Extra-P log also contains the fitting data in tabular form. Columns and entities are:
num processes / 1, mean time / s, median time / s

```
1  callpath: accu_comm_time
2    metric: Test
3      8.10E+01 Mean: 6.11E+00 Median: 6.11E+00
4      1.44E+02 Mean: 1.31E+01 Median: 1.31E+01
5      2.89E+02 Mean: 2.70E+01 Median: 2.69E+01
6      5.29E+02 Mean: 5.62E+01 Median: 5.60E+01
7      1.09E+03 Mean: 1.13E+02 Median: 1.13E+02
8      model: -2.37568+0.106594*(comm_size^1)
9      RSS: 7.26E+00
10     Adjusted R^2: 9.99E-01
11 callpath: accu_dotproduct
12   metric: Test
13     8.10E+01 Mean: 6.93E+00 Median: 6.90E+00
14     1.44E+02 Mean: 9.26E+00 Median: 9.25E+00
15     2.89E+02 Mean: 1.31E+01 Median: 1.31E+01
16     5.29E+02 Mean: 1.78E+01 Median: 1.78E+01
17     1.09E+03 Mean: 2.55E+01 Median: 2.55E+01
18     model: -0.0214981+0.773084*(comm_size^0.5)
19     RSS: 1.16E-04
20     Adjusted R^2: 1.00E+00
21 callpath: accu_eval_time
22   metric: Test
23     8.10E+01 Mean: 1.70E+04 Median: 1.70E+04
24     1.44E+02 Mean: 1.71E+04 Median: 1.71E+04
25     2.89E+02 Mean: 1.71E+04 Median: 1.70E+04
26     5.29E+02 Mean: 1.71E+04 Median: 1.71E+04
27     1.09E+03 Mean: 1.72E+04 Median: 1.72E+04
28     model: 17077.6
29     RSS: 7.54E+03
30     Adjusted R^2: 0.00E+00
```

```
31 callpath: accu_idle_bcast
32    metric: Test
33        8.10E+01 Mean: 4.53E-01 Median: 4.59E-01
34        1.44E+02 Mean: 1.09E+00 Median: 1.12E+00
35        2.89E+02 Mean: 3.50E+00 Median: 3.44E+00
36        5.29E+02 Mean: 6.74E+00 Median: 6.50E+00
37        1.09E+03 Mean: 1.69E+01 Median: 1.66E+01
38        model: -0.105609+0.00272605*(comm_size^1.25)
39        RSS: 1.72E-01
40        Adjusted R^2: 9.99E-01
41 callpath: accu_idle_eval
42    metric: Test
43        8.10E+01 Mean: 5.44E+01 Median: 5.37E+01
44        1.44E+02 Mean: 8.33E+01 Median: 5.47E+01
45        2.89E+02 Mean: 6.23E+01 Median: 5.79E+01
46        5.29E+02 Mean: 7.59E+01 Median: 7.58E+01
47        1.09E+03 Mean: 1.38E+02 Median: 1.64E+02
48        model: 66.5427+5.54734e-08*(comm_size^3)
49        RSS: 4.56E+02
50        Adjusted R^2: 8.60E-01
51 callpath: accu_idle_reading
52    metric: Test
53        8.10E+01 Mean: 3.51E+00 Median: 3.42E+00
54        1.44E+02 Mean: 7.22E+00 Median: 6.83E+00
55        2.89E+02 Mean: 1.92E+01 Median: 1.88E+01
56        5.29E+02 Mean: 7.90E+01 Median: 5.19E+01
57        1.09E+03 Mean: 1.05E+02 Median: 1.02E+02
58        model: -44.9403+4.66358*(comm_size^0.5)
59        RSS: 5.81E+02
60        Adjusted R^2: 9.09E-01
61 callpath: accu_idle_reductions
62    metric: Test
63        8.10E+01 Mean: 1.58E+00 Median: 1.61E+00
64        1.44E+02 Mean: 4.28E+00 Median: 4.30E+00
65        2.89E+02 Mean: 9.70E+00 Median: 9.60E+00
66        5.29E+02 Mean: 2.14E+01 Median: 2.13E+01
67        1.09E+03 Mean: 4.74E+01 Median: 4.76E+01
68        model: -0.33918+0.00436969*(comm_size^1)*log2^1(comm_size)
69        RSS: 8.78E-01
70        Adjusted R^2: 9.99E-01
71 callpath: accu_idle_writing
72    metric: Test
73        8.10E+01 Mean: 3.10E+01 Median: 2.80E+01
74        1.44E+02 Mean: 7.09E+01 Median: 6.51E+01
75        2.89E+02 Mean: 1.87E+02 Median: 1.82E+02
76        5.29E+02 Mean: 2.77E+02 Median: 2.67E+02
77        1.09E+03 Mean: 1.21E+03 Median: 7.48E+02
78        model: 60.4652+9.58279e-05*(comm_size^2)*log2^1(comm_size)
79        RSS: 5.57E+03
80        Adjusted R^2: 9.92E-01
81 callpath: accu_idletime
82    metric: Test
```

```
83          8.10E+01 Mean: 9.09E+01 Median: 8.64E+01
84          1.44E+02 Mean: 1.67E+02 Median: 1.57E+02
85          2.89E+02 Mean: 2.82E+02 Median: 2.77E+02
86          5.29E+02 Mean: 4.60E+02 Median: 4.62E+02
87          1.09E+03 Mean: 1.52E+03 Median: 1.10E+03
88          model: 104.092+0.0068053*(comm_size^1.75)
89          RSS: 4.62E+03
90          Adjusted R^2: 9.95E-01
91  callpath: accu_inputlen_reading
92      metric: Test
93          8.10E+01 Mean: 4.76E+00 Median: 9.37E-01
94          1.44E+02 Mean: 4.44E+00 Median: 5.35E+00
95          2.89E+02 Mean: 1.09E+01 Median: 1.05E+01
96          5.29E+02 Mean: 2.67E+01 Median: 3.07E+01
97          1.09E+03 Mean: 7.10E+01 Median: 4.76E+01
98          model: 2.20664+0.000610176*(comm_size^1.33333)*log2^1(comm_size)
99          RSS: 4.18E+00
100         Adjusted R^2: 9.98E-01
101 callpath: accu_io_time
102     metric: Test
103         8.10E+01 Mean: 4.35E+01 Median: 3.84E+01
104         1.44E+02 Mean: 8.97E+01 Median: 8.48E+01
105         2.89E+02 Mean: 2.30E+02 Median: 2.47E+02
106         5.29E+02 Mean: 3.98E+02 Median: 3.83E+02
107         1.09E+03 Mean: 1.42E+03 Median: 9.94E+02
108         model: 47.9264+0.00662936*(comm_size^1.75)
109         RSS: 4.01E+03
110         Adjusted R^2: 9.96E-01
111 callpath: accu_pure_reading
112     metric: Test
113         8.10E+01 Mean: 4.36E-01 Median: 4.26E-01
114         1.44E+02 Mean: 6.53E-01 Median: 6.19E-01
115         2.89E+02 Mean: 1.20E+00 Median: 1.17E+00
116         5.29E+02 Mean: 3.59E+00 Median: 2.35E+00
117         1.09E+03 Mean: 3.27E+00 Median: 3.18E+00
118         model: -5.61552+0.911702*log2^1(comm_size)
119         RSS: 1.56E+00
120         Adjusted R^2: 7.65E-01
121 callpath: accu_pure_result_comm
122     metric: Test
123         8.10E+01 Mean: 2.92E+00 Median: 2.90E+00
124         1.44E+02 Mean: 4.61E+00 Median: 4.60E+00
125         2.89E+02 Mean: 7.98E+00 Median: 7.98E+00
126         5.29E+02 Mean: 1.36E+01 Median: 1.36E+01
127         1.09E+03 Mean: 2.29E+01 Median: 2.29E+01
128         model: -0.467172+0.123781*(comm_size^0.75)
129         RSS: 2.20E-01
130         Adjusted R^2: 9.99E-01
131 callpath: accu_pure_writing
132     metric: Test
133         8.10E+01 Mean: 3.86E+00 Median: 3.49E+00
134         1.44E+02 Mean: 6.42E+00 Median: 5.89E+00
```

```
135            2.89E+02  Mean:  1.15E+01  Median:  1.13E+01
136            5.29E+02  Mean:  1.19E+01  Median:  1.19E+01
137            1.09E+03  Mean:  3.41E+01  Median:  2.22E+01
138            model:  5.20517+0.000248423*(comm_size^1.66667)
139            RSS:  1.69E+01
140            Adjusted  R^2:  9.61E−01
141   callpath:  accu_runtime
142       metric:  Test
143            8.10E+01  Mean:  1.71E+04  Median:  1.71E+04
144            1.44E+02  Mean:  1.72E+04  Median:  1.72E+04
145            2.89E+02  Mean:  1.74E+04  Median:  1.74E+04
146            5.29E+02  Mean:  1.76E+04  Median:  1.76E+04
147            1.09E+03  Mean:  1.89E+04  Median:  1.85E+04
148            model:  17147.5+0.0148578*(comm_size^1.66667)
149            RSS:  5.61E+03
150            Adjusted  R^2:  9.96E−01
```

8.4 Images

Figure 36 Accumulate idle-times after evaluations in strongscaling: automated analysis of logged timings with Extra-P for a strong-scaling experiment with a 2.93×10^6 `sample` input series, as explained in section 6.3.2