A similarity study of I/O traces via string kernels

Article

Accepted Version

Torres, R., Kunkel, J. M., Dolz, M. F. and Ludwig, T. (2018) A similarity study of I/O traces via string kernels. The Journal of Supercomputing. ISSN 0920-8542 doi: https://doi.org/10.1007/s11227-018-2471-x Available at http://centaur.reading.ac.uk/78027/

Journal of Supercomputing manuscript No.
(will be inserted by the editor)

A Similarity Study of I/O Traces via String Kernels

**Raul Torres · Julian Kunkel ·
Manuel F. Dolz · Thomas Ludwig**

Received: date / Accepted: date

Abstract Understanding I/O for data-intense applications is the foundation for the optimization of these applications. The classification of the applications according to the expressed I/O access pattern eases the analysis. An access pattern can be seen as fingerprint of an application. In this paper, we address the classification of traces. Firstly, we convert them first into a weighted string representation. Due to the fact that string objects can be easily compared using Kernel Methods, we explore their use for fingerprinting I/O patterns. To improve accuracy, we propose a novel string kernel function called *Kast2 Spectrum Kernel*. The similarity matrices, obtained after applying the mentioned kernel over a set of examples from a real application, were analyzed using Kernel Principal Component Analysis (Kernel PCA) and Hierarchical Clustering. The evaluation showed that two out of four I/O access pattern groups were completely identified, while the other two groups conformed a single cluster due to the intrinsic similarity of their members. The proposed strategy can be promisingly applied to other similarity problems involving tree-like structured data.

Keywords Kernel Functions · Kast2 Spectrum Kernel · I/O Access Pattern Comparison · String Kernels

Raul Torres, Thomas Ludwig
Department of Informatics, Universität Hamburg, 20146–Hamburg, Germany
E-mail: raul.torres@informatik.uni-hamburg.de, ludwig@dkrz.de

Julian M. Kunkel
Department of Computer Science, University of Reading, RG6 6AY–Reading, UK
E-mail: j.m.kunkel@reading.ac.uk

Manuel F. Dolz
Department of Computer Science, Universidad Carlos III de Madrid, 28911–Leganés, Spain
E-mail: mdolz@inf.uc3m.es

1 Introduction

The identification and analysis of I/O access patterns is important in High Performance Computing because it helps, not only to understand the impact factors on the underlying Parallel File System, but also to design better ways of organizing I/O operations.

In order to understand the correlation of a collection of patterns, two requirements have to be met: *i)* a proper representation able to abstract the relevant features of each pattern and *ii)* an appropriate strategy to find similarities or dissimilarities between the data in this new representation. To tackle *i)* this paper uses a string conversion technique previously proposed by the authors [1]. In order to tackle *ii)* the obtained strings are compared with a novel string kernel function called *Kast2 Spectrum Kernel*. This ultimately allows to determine a similarity score between different access patterns.

This paper is organized as follows: in Section 2, the basic foundations of parallel I/O and string kernels are presented. Section 3 explains the rational behind the proposed kernel function. The evaluation of the approach is conducted in Section 4. Section 5 revisits some related works in the area. Finally, Section 6 summarizes the results and details possible future paths for the current research efforts.

2 Background

In this section we describe the main topics related to this research: *i)* parallel file systems; *ii)* kernel methods for similarity search.

2.1 Parallel file systems

According to Kunkel [2], Parallel File Systems are minded for accessing files in a simultaneous, concurrent and efficient way. In order to achieve highest performance, the contents of a file are usually scattered among different I/O servers. Parallel File Systems should provide, among other capabilities, persistence, consistence, performance, manageability, scalability, fault-tolerance and availability. Different approaches can be used to analyze the performance of a Parallel File System. Investigating the mere performance, however, is not sufficient as optimal performance is a function of the exhibited access pattern. This makes it difficult for an observer to assess any measured performance. A typical strategy is to relate performance of an application with a similarly behaving application for which we know how well it behaves – like benchmarks, for example. Likewise, we may be interested to identify well-behaving or ill-behaving applications. Therefore, finding patterns via fingerprints inside I/O traces is an important use case.

I/O Access Patterns. I/O access patterns depict the behavior of data access over a period of time; hence, they can be used to assess performance of an I/O system. The following properties characterize an access pattern: access granularity, randomness, concurrency, load balance, access type and predictability.

Liu et al. [3] added three additional characteristics seen on supercomputing I/O patterns: burstiness, periodicity and repeatability.

2.2 Kernel methods for similarity search

As stated by Kung [4], a typical machine learning system consists of two subsystems: the feature extraction and the clustering/classifier subsystems. The feature extraction subsystem performs the process of conversion of raw data to a meaningful representation. The clustering/classifier subsystem makes reference to the strategy used to distill information from the new representation. There is group of algorithms, among the constellation of machine learning techniques, that have been successfully applied in structured data problems: they are called Kernel Methods [5]. This group of algorithms can detect stable patterns robustly and efficiently from a finite data sample; their idea is to embed the original data into a space where linear relations manifest as patterns. These methods have been successfully applied in problems with structured data types like trees and strings [6]. Kernel methods follow the mentioned two-stages strategy: firstly, a mapping is made by the Kernel Function, which depends on the specific data type and domain knowledge, and secondly, a general purpose and robust kernel learning algorithm is applied to find the linear relationships in the induced feature space. The stage of construction of the kernel function can be characterized as follows:

- Original data items are embedded into a vector space called *feature space*.
- The images of data in the feature space have linear relations.
- The learning algorithm does not need to know the coordinates of the feature space data; the pairwise inner products are enough.
- These inner products can be calculated efficiently using a kernel function.

The inner products conform the *kernel matrix*, which is the only piece of information that the learning algorithms need, in order to extract meaningful information. In this work we used two algorithms: Hierarchical Clustering [7] and Kernel Principal Component Analysis (Kernel PCA) [8].

String Kernels. Usually, data is delivered as a collection of attribute-value tuples; the widely used Polynomial and Gaussian Kernels Functions are ideal for this kind of representation. But in the case of structured data like trees and strings, the design of kernel functions becomes more complex. Despite this complexity, some solutions have been proposed, for example, Convolution Kernels [9–11]. Strings kernels are explained in a comprehensive way in [12]. They check for the number of shared substrings among a collection of strings. These substrings must comply with certain weighting factors, which produces different kernel functions. The bag-of-characters kernel, for example, only takes into account single-character matching. The bag-of-words kernel searches for shared words among strings. The k-spectrum kernel [13] only counts sub-strings of length k. The k-blended spectrum kernel[5] only counts sub-strings whose length is not major than a given number k.

3 Methodology

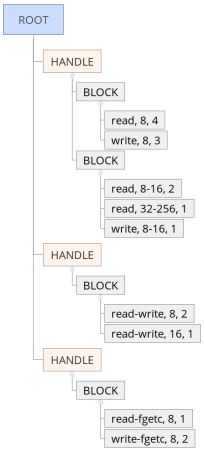
In this section, we describe our method for converting I/O traces into weighted strings. We also present a novel kernel function to compare the resulting strings.

3.1 Conversion of I/O traces into strings

The strategy used for converting I/O traces into strings was proposed by the authors in a previous article [1]. In a first stage, the I/O traces are converted into trees as follows:

- At the top level, an imaginary root node groups all the operations of a single I/O access pattern file.
- At the next level, imaginary nodes group all the operations belonging to the same file handle.
- At the following level, imaginary nodes group all the operations found between an `open` operation and its corresponding `close` operation.
- At the bottom level, operations are given nodes, except for `open` and `close`.

In a second stage (see Fig. 1), the resulting trees are traversed in pre-order and each node's properties are extracted. For each node of the tree, a token in the string is created. A token is compound by a literal part and a weight value. For leaf nodes the literal part is formed with the name of the operation and the number of bytes enclosed by `[ ]` while their weight corresponds to the number of repetitions. To preserve information about the tree structure, we introduced a new token that does not correspond to any node but gives a notion of distance between nodes: the `[LEVEL_UP]` token represents the change to an upper level when doing the pre-order traversal and its weight is simply the amount of levels jumped until the next new node is found.



(a) Abstract Syntax Tree.

Tokens	Repetitions
[ROOT]	1
[HANDLE]	1
[BLOCK]	1
[read_8]	4
[LEVEL_UP]	1
[write_8]	3
[LEVEL_UP]	2
[write_8-16]	2
[LEVEL_UP]	1
[write_32-256]	1
[LEVEL_UP]	1
[write_8-16]	1
[LEVEL_UP]	3
[HANDLE]	1
[BLOCK]	1
[read-write_8]	2
[LEVEL_UP]	1
[read-write_16]	1
[LEVEL_UP]	3
[HANDLE]	1
[BLOCK]	1
[read-fgetc_8]	1
[LEVEL_UP]	1
[write-fgetc_8]	2
[LEVEL_UP]	4

(b) Extracted tokens.

Fig. 1 Creation of a string of tokens from a tree.

3.2 Comparison of strings: the Kast2 Spectrum Kernel

Once data structures are converted into weighted strings, they can be easily compared using a string kernel function. In this study, a novel string kernel function is proposed: the *kast2 spectrum kernel*. In theory, the number of different tokens that can compound a string is infinite. In practice, this number is limited to the namespace of a particular domain. For the case of I/O traces, it is limited by the I/O operations names and the number of bytes related to each operation. Still, the number of tokens can be very high. In an hypothetical feature space, where every string is characterized by the presence or absence of each possible token with each possible weight, the number of features is still infinite. However, in practice, for a single string, most of the features of this hypothetical space are zero-valued. This is a fact that eases the creation of a feasible kernel function. This way, the new embedding space has a finite and small number of features, that corresponds to the actual tokens that exist in a set of samples. For two weighted strings A and B, the kernel here proposed must follow the conditions given below:

1. The user must specify a minimum weight or "cut weight" value as parameter.
2. The aim is to find the longest matching substrings of A and B, whose weights are greater than or equal to the cut weight. They are called *valid matching substrings*. Invalid matching substrings have a weight value that is smaller than the cut weight, and are hence ignored.
3. A *valid matching substring* can appear more than once in each string.
4. A *valid matching substring* must not be a substring of another *valid matching substring* in at least one of the original strings.

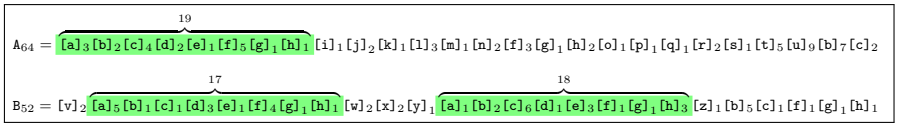


Fig. 2 S_1 is the largest substring found on both examples.

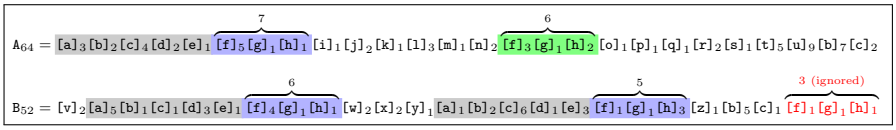


Fig. 3 S_2 appears once as an independent case.

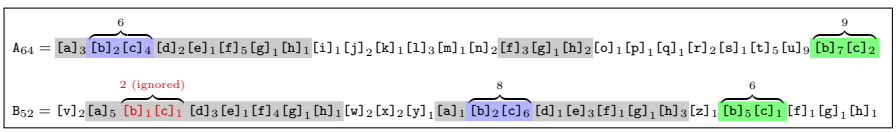


Fig. 4 S_3 appears twice as an independent case.

As an example, consider the strings A and B of Fig. 2, and a cut weight value of 4. The first *valid matching substring* with the longest size ($S1$) is found once in A and twice in B (see Fig. 2). The second longest *valid matching substring* ($S2$) is found twice in A and twice in B (see Fig. 3). This substring appears at least once as an independent substring in one of the strings, hence complying with condition 4. Notice that an extra occurrence is ignored because its weight is smaller than 4. The last and shortest *valid matching substring* ($S3$) is found twice in A and twice in B (see Fig. 4). As the substring appears as an independent case in both strings, it complies with condition 4. Here also an extra occurrence is ignored due to a smaller weight. The *kast2 spectrum kernel* has the following definition:

- Only the weights of the independent *valid matching substrings* are taken into account to build a partial feature value, which is the summation of these weights.
- If the string does not present an independent occurrence of a particular *valid matching substring*, no weight value is taken into account.
- Only a single feature value is created for each string A and B, which corresponds to the summation of all partial feature values.
- A penalization value is introduced and it corresponds to the number of effective segments reduced by one; an effective segment is an instance of an independent *valid matching substring*. this value is subtracted from the feature value; the reduction by one prevents exact matching strings from being penalized.
- The kernel value corresponds to the product of the feature values after penalization.

Example. Let A and B be the same strings from previous examples (see Fig. 2). The function $weight\_k2_{w \geq n}(S)_A$ returns, either:

- the summation of the weights of all the independent matching instances of S in A whose weight is greater than or equal to n,
- 0, if there are no independent substrings.

For a cut weight of 4 ($n = 4$), the respective weights of each partial feature in A are calculated with:

$$weight\_k2_{w \geq 4}(S1)_A = 19 \tag{1}$$

$$weight\_k2_{w \geq 4}(S2)_A = 6 \tag{2}$$

$$weight\_k2_{w \geq 4}(S3)_A = 9 \tag{3}$$

The only feature of A is the summation of the previous weights:

$$f2_{w \geq 4}(A) = 19 + 6 + 9 = 34 \tag{4}$$

The penalization value in this case is 2:

$$p2_{w \geq 4}(A) = 34 - 2 = 32 \tag{5}$$

Notice in Fig. 3 that $S2$ does not appear as an independent *valid matching substring* in B. Hence, the feature value is set to 0 (see Eq. 7). The respective weights of each substring in B are:

$$weight\_k2_{w\geq 4}(S1)_B = 17 + 18 = 35 \tag{6}$$

$$weight\_k2_{w\geq 4}(S2)_B = 0 \tag{7}$$

$$weight\_k2_{w\geq 4}(S3)_B = 6 \tag{8}$$

Here too, the only feature of B is the summation of the previous weights:

$$f2_{w\geq 4}(B) = 35 + 6 = 41 \tag{9}$$

For this case, the penalization value is also 2:

$$p2_{w\geq 4}(B) = 41 - 2 = 39 \tag{10}$$

The function $k2_{w\geq n}(A, B)$ returns the evaluation of the kernel value between A and B; this is no more than the product of these two values:

$$k2_{w\geq 4}(A, B) = \langle p2_{w\geq 4}(A), p2_{w\geq 4}(B)\rangle = 1248 \tag{11}$$

The function $\bar{k2}_{w\geq n}(A, B)$ is the normalized version of the kernel. A further normalization step using the weights of each string can be applied:

$$\bar{k2}_{w\geq 4}(A, B) = \frac{k2_{w\geq 4}(A, B)}{\sqrt{k2_{w\geq 4}(A, A) \times k2_{w\geq 4}(B, B)}} = \frac{k2_{w\geq 4}(A, B)}{weight\_k2_{w\geq 4}(A) \times weight\_k2_{w\geq 4}(B)} \tag{12}$$

$$\bar{k2}_{w\geq 4}(A, B) = \frac{1248}{64 \times 52} = \frac{1248}{3328} \approx 0.375 \tag{13}$$

In this case, the kernel emits a similarity score of 37.5 % between A and B.

4 Experimental evaluation

The experimental evaluation was designed to assess the capabilities of the proposed kernel over a set of examples from a real application. These capabilities were analyzed using Kernel PCA and Hierarchical Clustering.

4.1 Configuration

It was on the interest of this work to study the suitability of the proposed strategy to find similarities among four distinct classes of I/O access patterns, which have been obtained from two different parallel I/O benchmarks:

- *The IOR HPC Benchmark:* It is used for benchmarking parallel file systems that use POSIX, MPIIO, or HDF5 interfaces [15].
- *The FLASH I/O Benchmark:* It measures the performance of the FLASH parallel HDF5 output. FLASH is scientific tool for modeling astrophysical thermonuclear flashes [16].

The I/O traces were organized in the following classes of storage access:

- *Class A (Flash I/O):* 10 traces. Characterized for containing contiguous `write` operations with diverse byte values that are not present in the other classes.
- *Class B (Random I/O):* 4 traces. These ones present `lseek` operations not seen elsewhere.
- *Classes C and D (Sequential I/O):* 4 traces each. 8 in total. Classes C and D do not have any remarkable difference among them, a fact that has been confirmed by experimentation. However, they come from different runs, that is why we present them here separately.

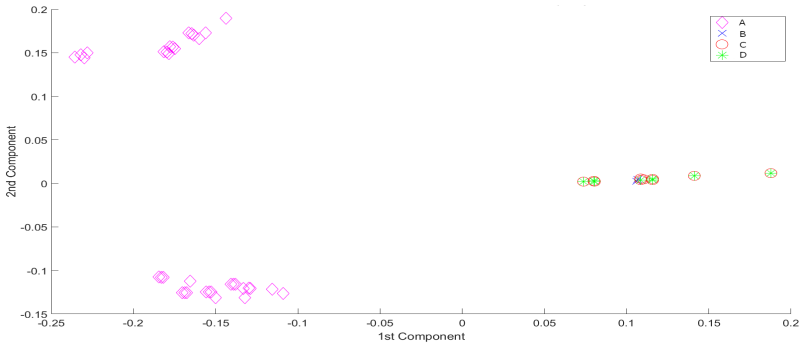
For each pattern four additional synthetic copies were created. Such copies introduced small mutations on the pattern; the idea behind these mutations was the need to create access patterns that were, in theory, closer to a determined example than the rest of the category members. Thus, from 22 examples we ended up with 110 samples, distributed as follows: (A) 50 examples, (B) 20 examples, (C) 20 examples and (D) 20 examples. Each access pattern was converted to the proposed string representations. The proposed *kast2 spectrum kernel* function was applied to them, and the results were compared to the ones presented in a previous publication from the authors [1], using the *blended spectrum kernel* and the *kast spectrum kernel* as baseline kernels. The selected cut weight values were the following: $\{2^1, 2^2, ..., 2^n\} : n = 10$. If the matrices presented negative eigenvalues, they were replaced by zero and the matrices rebuilt. All the similarity matrices were analyzed with both Kernel PCA and Hierarchical Clustering, the latest using the simple linkage method.

4.2 Baseline kernel 1: Blended spectrum kernel

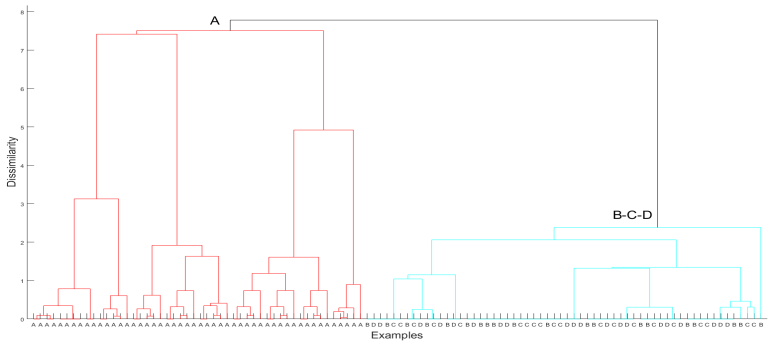
In a previous work from the authors [1], the *blended spectrum kernel* was able to separate only one category of I/O patterns (Flash I/O) from the rest when analyzed with Hierarchical Clustering (see Figure 5b). Moreover, with Kernel PCA, it divided the Flash I/O patterns into two parts (see Figure 5a).

4.3 Baseline kernel 2: Kast spectrum kernel

In the same work it was shown that the *kast spectrum kernel*, proposed by the authors, was able to detect 3 clusters with no misplaced examples (see Figures 6a and 6b). While Flash I/O (A) and Random POSIX I/O (B) were separated independently, Normal I/O and Random Access I/O (C-D) were placed on the same group. This corresponded to the structure of each category: (A) examples contained contiguous `write` operations with different byte values that were not present in the other categories. (B) examples contained `lseek` operations not seen elsewhere. (C) and (D) shared the same pattern. However, this clustering could only be obtained when the cut weight was small.



(a) Kernel PCA.



(b) Hierarchical clustering.

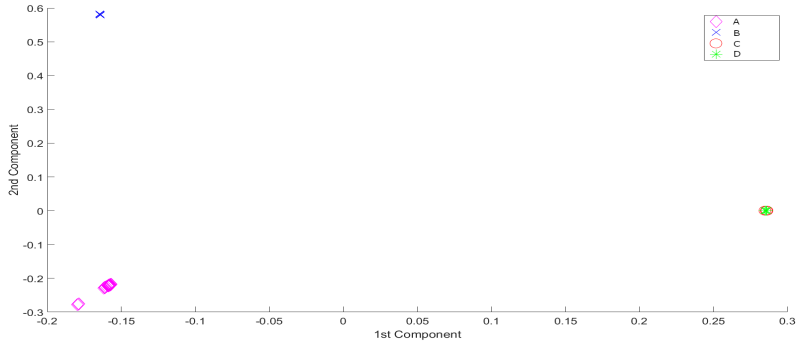
Fig. 5 Blended Spectrum Kernel using byte information (cut weight = 2).

4.4 Proposed kernel: Kast2 spectrum kernel

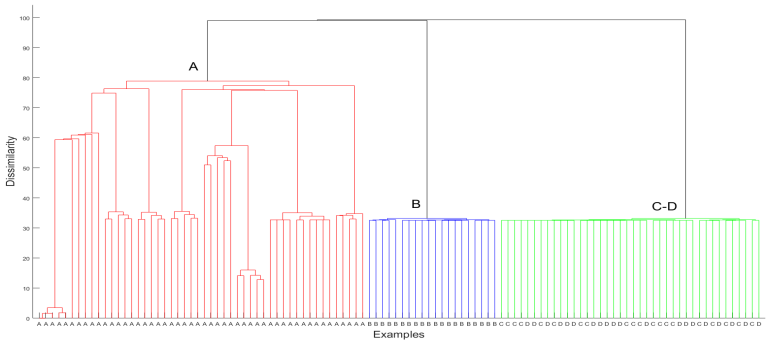
For the new kernel here proposed, the results outperformed both the ones from the baseline kernels, as the selection of the cut weight did not have a significant effect on the inter-cluster and the intra-cluster distances (see Figures 7b and 7a). This stability is an advantage with respect to all other kernels analyzed. The usual clusters were found: Flash I/O (A), POSIX I/O (B), and Normal and Random I/O (C-D).

5 Related work

Kluge [14] proposed an intermediate representation of I/O events from High Performance Computing (HPC) applications as a Directed Acyclic Graph (DAG). In this DAG vertices are used to represent events while edges are used to depict the chronological order of the events. Kluge also proposed a redundancy elimination step where adjacent synchronization vertices can be merged in a single one. Madhyastha et al. [17] applied two supervised learning algorithms to classify Parallel I/O access patterns: a feed forward neural network and a hidden Markov models based approach. Both strategies require



(a) Kernel PCA.



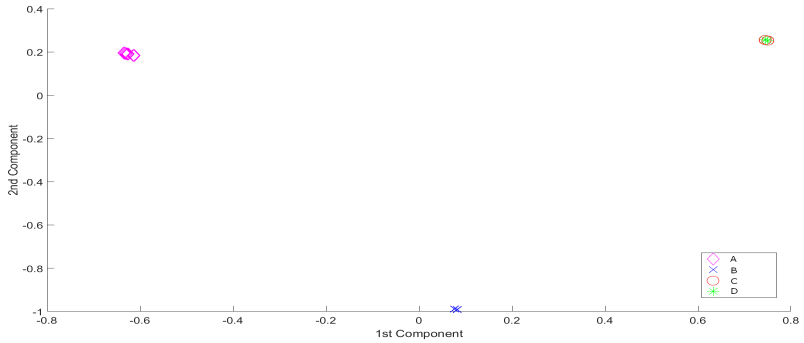
(b) Hierarchical clustering.

Fig. 6 Kast Spectrum Kernel using byte information (cut weight = 2).

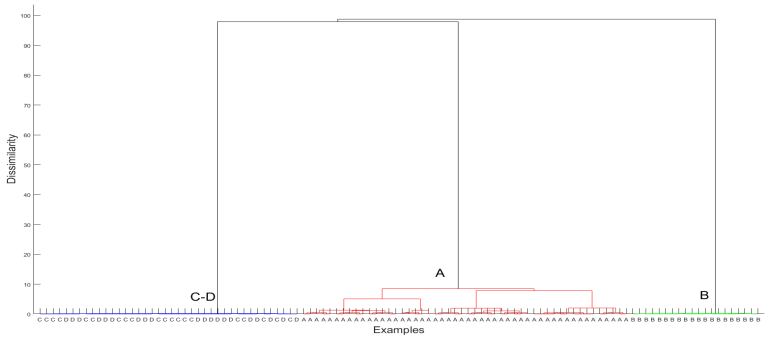
training with previously labeled examples. Behzad et al. [18] proposed and I/O auto tuning framework that extracts the patterns from an application and searches for a match on a database of previously known pattern models. If there is a match, the associated model is adopted on the fly during the execution of the application. A different abstraction approach was made by Liu et al. [3]. They used the I/O bursts registered on noisy server-side logs of an application as a signature to find similarities between I/O samples. The final signature is a 2D grid called CLIQUE [19] that relates a correlation coefficient with time. Because the signature extraction was made over log files there was zero overhead in the application performance. Koller and Rangaswami [20] used disk static similarity and workload static similarity at the block level to analyze the performance of concurrent applications of the same file system. Unfortunately, we could not find suitable studies on I/O pattern similarity with kernel methods for comparing our results.

6 Conclusions and future work

In this paper we showed how the I/O traces of a parallel program can be used to find access patterns. A set of traces, taken from a real parallel appli-



(a) Kernel PCA.



(b) Hierarchical clustering.

Fig. 7 Kast2 Spectrum Kernel using byte information (cut weight = 2).

cation where 4 distinct patterns were present, were represented as weighted strings and afterwards compared using a novel kernel function proposed by the authors. The *kast2 spectrum kernel* emitted a similarity matrix between examples that was later analyzed with Kernel PCA and Hierarchical Clustering. These algorithms showed a consistent formation of three groups according to the expected patterns with no misplaced examples. These findings clearly confirm that both the previously proposed string representation and the novel comparison method are suitable to find I/O access patterns in a parallel application. However, due to the fact that the proposed string representation is independent from the domain, it can also be used to compare I/O access of a sequential program. Future efforts of this project will focus on the comparison of the intermediate representation delivered by the LLVM Compiler Infrastructure.

Acknowledgements

Raul Torres would like to acknowledge the financial support from the Colombian Administrative Department of Science, Technology and Innovation (Colciencias) as well as the mathematical advisory received from Ruslan Krenzler.

References

1. R. Torres, J.M. Kunkel, M.F. Dolz and T. Ludwig. A Novel String Representation and Kernel Function for the Comparison of I/O Access Patterns. Parallel Computing Technologies: Lecture Notes in Computer Science, Vol 10421. Springer, Cham (2017)
2. Kunkel J. M., Simulating parallel programs on application and system level. Computer Science –Research and Development. 28(2), pp. 167–174 (2012)
3. Liu Y., Gunasekaran R., Ma XS. and Vazhkudai S. S.: Automatic Identification of Application I/O Signatures from Noisy Server-Side Traces. In Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14). pp. 213–228, Santa Clara (2014)
4. Kung S. Y.: Kernel Methods and Machine Learning. Cambridge University Press. (2014)
5. Shawe–Taylor J., and Cristianini N.: Kernel Methods for Pattern Analysis. Cambridge University Press New York, New York (2004)
6. BakIr G., Hofmann T., Schölkopf B., Smola A. J., Taskar B., and Vishwanathan S.V.N: Predicting Structured Data. The MIT Press (2007)
7. Hastie T., Tibshirani R., and Friedman J.: The Elements of Statistical Learning - Data Mining, Inference, Springer Series in Statistics, Springer-Verlag, New York, (2009)
8. Schölkopf B., Smola A., and Müller K-R.: Kernel principal component analysis. In International Conference on Artificial Neural Networks, ICANN 1997: Artificial Neural Networks – ICANN'97, pp. 583–588, (1997)
9. Gärtner, T., Lloyd J. W., and Flach, P. A.: Kernels for structured data. In ILP'02 Proceedings of the 12th international conference on Inductive logic programming. pp. 66–83, Sidney (2002)
10. Gärtner, T., Lloyd J. W., and Flach, P. A.: Kernels and Distances for Structured Data. Machine Learning. 57(3), 205–232, (2004)
11. Haussler D.: Convolution Kernels on Discrete Structures. Technical Report. University of California at Santa Cruz, (1999)
12. Vishwanathan S. V. N., and Smola A. J.: Fast Kernels for String and Tree Matching. In Advances in Neural Information Processing Systems 15, pp. 569–576. (2003)
13. Leslie C., Eskin E., and Noble W.S.: The spectrum kernel: a string kernel for SVM protein classification. In Proceedings of the Pacific Symposium on Biocomputing. Vol. 7. pp. 566–575 (2002)
14. Kluge M.: Comparison and End-to-End Performance Analysis of Parallel Filesystems. PhD Thesis Dissertation. Technische Universität Dresden. (2011)
15. Loewe W., McLarty T., and Morrone C.: IOR Benchmark. (2012)
16. Fryxell b., Olson K., Ricker P., Timmes F. X., Zingale M., Lamb D. Q., MacNeice P., Rosner R., Truran J. W., and Tufo H.: FLASH: An Adaptive Mesh Hydrodynamics Code for Modeling Astrophysical Thermonuclear Flashes. The Astrophysical Journal Supplement Series. 131(1), 273, (2000)
17. Madhyastha T. M., and Reed D. A.: Learning to Classify Parallel Input/Output Access Patterns. IEEE Trans. on Parallel and Distributed Systems. 13(8), pp. 802–813. (2002)
18. Behzad B., Byna S., Prabhat and Snir, M.: Pattern-driven Parallel I/O Tuning. In Proceedings of the 10th Parallel Data Storage Workshop, pp. 43–48. Austin, Texas (2015)
19. Agrawal R., Gehrke J., Gunopulos D., and Raghavan P.: Automatic Subspace Clustering of High Dimensional Data for Data Mining Applications. In: SIGMOD '98 Proceedings of the 1998 ACM SIGMOD international conference on Management of data, pp. 94–105. Seattle (1998)
20. Koller R., and Rangaswami R.: I/O Deduplication: Utilizing content similarity to improve I/O performance. ACM Transactions on Storage. 6(3), pp. 13:1–13:26. (2010)