A Novel String Representation and Kernel Function for the Comparison of I/O Access Patterns

Raul Torres(0000-0001-6050-1227)*, Julian Kunkel(0000-0002-6915-1179), Manuel F. Dolz(0000-0001-9466-3398), and Thomas Ludwig

Universität Hamburg, Scientific Computing Research Group, Hamburg, Germany
raul.torres@informatik.uni-hamburg.de

Abstract. Parallel I/O access patterns act as fingerprints of a parallel program. In order to extract meaningful information from these patterns, they have to be represented appropriately. Due to the fact that string objects can be easily compared using Kernel Methods, a conversion to a weighted string representation is proposed in this paper, together with a novel string kernel function called Kast 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 2 out of 4 I/O access pattern groups were completely identified, while the other 2 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, Kast spectrum kernel, I/O access pattern comparison, Kernel PCA

1 Introduction

I/O access patterns act as fingerprints of an application. The identification and analysis of these 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: a) a proper representation able to abstract the relevant features of each pattern and b) an appropriate strategy to find similarities or dissimilarities between the data in this new representation. To tackle a) this paper proposes a two-stage string conversion technique for access patterns. The first stage transforms the data and reflects the containment relationships of the pattern in a tree-like data structure. The second stage flattens the resulting tree and simplifies the representation in a weighted string. In order to tackle b) these weighted

* raul.torres@informatik.uni-hamburg.de

The final publication is available at Springer via http://dx.doi.org/10.1007/978-3-319-62932-2_48

strings are compared with a novel string kernel function called Kast Spectrum Kernel.

2 Background

2.1 Parallel File Systems

Generalities Parallel File Systems [1] are minded for accessing files in a simultaneous, concurrent and efficient way. The contents of a file are usually scattered among different I/O subsystems in order to take advantage of the highest local performance of each subsystem. These systems should provide, among other capabilities, persistence, consistence, performance, and manageability. Other desired features might include: scalability, fault-tolerance and availability. Different approaches can be used to analyze the performance of a Parallel File System. Checking the patterns of the I/O traces is among the most commonly used ones.

I/O Access Patterns I/O access patterns depict the behavior of disk access over a period of time. They can be used to determine the overall performance of an I/O system. It is possible to characterize them by the following properties: access granularity, randomness, concurrency, load balance, access type and predictability. Liu et al. [2] mentioned three additional features seen on supercomputing I/O patterns: burstiness, periodicity and repeatability.

2.2 Kernel Methods for Similarity Search

As stated in [3], a typical machine learning systems consists of two subsystems: the feature extraction and clustering/classifier subsystems. On the one hand, the feature extraction subsystem performs the process of conversion of raw data to a meaningful representation. On the other hand, 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. Kernel Methods are well documented in the book of Shawe-Taylor and Cristianini [4]. This group of algorithms are strong enough to detect stable patterns robustly and efficiently from a finite data sample; basically, the 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 [5]. Kernel methods follow the mentioned two-stages strategy: first, a mapping is made by the Kernel Function, which depends on the specific data type and domain knowledge. Second, 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 in an efficient way using a kernel function.

The inner products between the training examples conform the *kernel matrix*. The learning algorithms are independent from the kernel function and need only the kernel matrix to extract meaningful information from the data. In this work we used two algorithms: Hierarchical Clustering [6] and Kernel Principal Component Analysis (Kernel PCA) [7].

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 for 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 [8–10]. Strings kernels are explained in a comprehensive way in [11]. They basically 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 only takes into account single-character matching. The bag-of-words kernel searches for shared words among strings. The k-spectrum kernel[12] only counts sub-strings of length k. The k-blended spectrum kernel[4] only counts sub-strings which length are less or equal to a given number k.

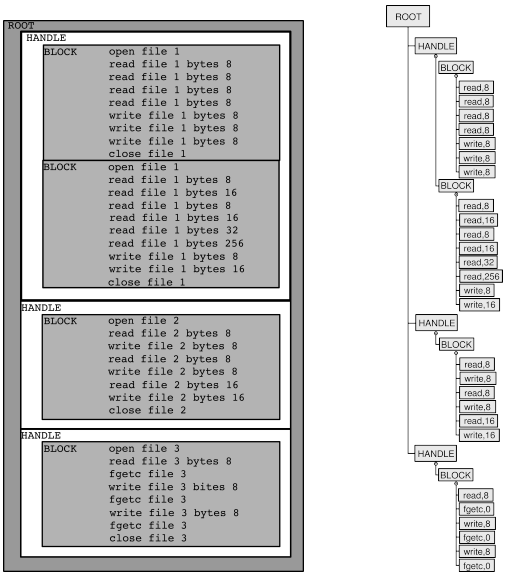
3 Methodology

3.1 Creating Strings from I/O Access Patterns

The I/O access pattern files are plain text files where each line corresponds to an operation. Some of these operations are negligible and hence ignored (e.g. **fileno**, **nmap** and **fscanf**). Some other operations keep information of the number of bytes involved on it. The proposed string representation can either use or ignore such byte information (ignoring is made by assuming all byte values are zero), which means that two different type of strings can be generated from a single I/O access pattern. Operations in the I/O access pattern are registered chronologically; with several file handles acting at the same time it is not always possible that all the operations belonging to the same file handle could have been written contiguously. For that reason the patterns are first converted into trees. Trees are ideal data structures for representing containment relationships between objects.

From I/O Access Patterns to Trees The trees that we use in this paper will have the following levels: The *ROOT* level, the *HANDLE* level, the *BLOCK* level and the operation level (See Figure 1):

– At the highest level, an imaginary root node groups all the operations of a single I/O access pattern file. Such node is represented as *ROOT*.
– At the second level, imaginary nodes group all the operations belonging to the same file handle. Such nodes are represented as *HANDLE*.
– At the third level, imaginary nodes group all the operations found between an **open** operation and its corresponding **close** operation. Such nodes are represented as *BLOCK*.
– At the deepest level, operations are given nodes, except for **open** and **close**, because the *BLOCK* node already plays the role of a delimiter.



(a) Access pattern (b) Resulting tree

Fig. 1. Conversion of a plain text I/O access pattern into a tree

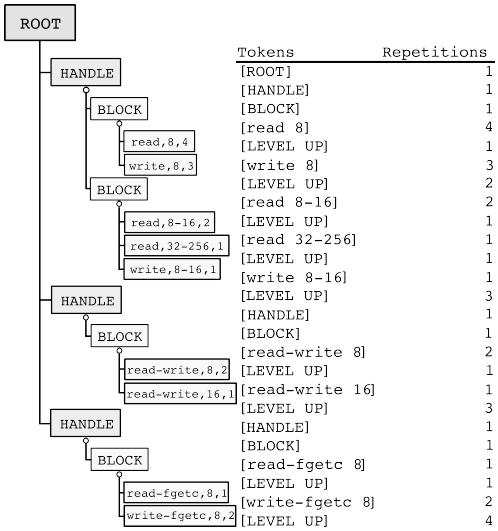
In order to save space, a set of consecutive operation nodes on the same block can be expressed as a single node when they present some simple patterns. A similar approach was applied by Kluge [13]. The resulting node will have an additional field that stores of the number of repetitions. This compression step is based on the following transformations, which are performed in the given order:

– Consecutive operations with the same name and the same number of bytes are simplified to a single operation with the same information. E.g. a **read** operation inside a loop reading a file n bytes per iteration.

- Consecutive operations with the same name but different number of bytes are simplified to a single operation with the same name. The new byte value is a combination of both previous byte numbers. E.g. initializing in a loop an array of C structures compound of a 2-bytes integer and a 4-bytes integer will need a **read** operation extracting two bytes first and another **read** extracting four bytes afterwards.
- Consecutive operations with different name but same number of bytes are simplified to a single operations with the same number of bytes. The new operation name is a combination of both previous names. E.g., a series of interlaced **read** and **write** operations with the same number of bytes might indicate a tacit copy operation.
- Consecutive operations with different name and different number of bytes but with one operation having 0 as number of bytes are simplified to a single operation with the non-zero value as the number of bytes. The new operation name is a combination of both previous names. E.g. inside a loop an **lseek** operation moves the pointer in the file descriptor and a **write** operation records the information there.

The previous steps are repeated once again to capture higher level patterns. Some of the operations (e.g. **read**, **write**) have a memory address associated to them. If this values would be taken into account, the compression step would be more precise to capture related operations e.g a copy operation. However, the degree of compression would be reduced. The main interest of this research is to use patterns for determining in an efficient way how similar a collection of I/O traces are, not to break down the pattern and try to understand the underlying structure of it. For this reason, the memory addresses are ignored completely.

From Trees to Strings Once the tree is compacted, the string representation can be built. The process is straightforward (See Figure 3.1). The tree is traversed in pre-order and each node properties are extracted; each node of the tree corresponds to a token in the string. 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. *ROOT*, *HANDLE* and *BLOCK* nodes are translated as *[ROOT]*, *[HANDLE]* and *[BLOCK]* respectively; their weight is always 1. To preserve information about the tree structure, we introduced a new token that does not correspond to any node but give a notion of distance between nodes. The rational of this design corresponds to the future application of this representation in more complex structures like Abstract Syntax Trees (ASTs). The *[LEVEL_UP]* token represents the change to an upper level when doing the pre-order traversal. Its weight is simply the amount of levels jumped until the next new node is found. Notice that there is no need for a token to indicate a change to a lower level, due to the fact that in the pre-order traversal the number of levels jumped from a parent to a child is always 1, which is implicitly expressed when two tokens are written one after the other.



Tokens	Repetitions
[ROOT]	1
[HANDLE]	1
[BLOCK]	1
[read 8]	4
[LEVEL UP]	1
[write 8]	3
[LEVEL UP]	2
[read 8-16]	2
[LEVEL UP]	1
[read 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

(a) Compacted tree (b) Extracted tokens

Fig. 2. Creation of a string of tokens from a tree

3.2 Comparing Strings: the Kernel Function

The basic idea is to create a comparison measure for strings conformed by weighted tokens. In theory, the number of different tokens is infinite. In practice, the number of different tokens can be limited to the I/O operations on a program and the number of bytes related to each operation; still, the number is high. In order to define a proper similarity measure, it is necessary to define first some important concepts:

- A weighted string is a set of consecutive weighted tokens (from here on out referred simply as strings and tokens).
- A substring is a string that is fully contained by another string.
- The weight of a string is the summation of the weights of its tokens.

It is easy to infer here that the number of possible strings is also infinite. 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. In this work the *Kast Spectrum Kernel* is proposed. In this kernel, some conditions have to be met to build the new embedding space:

- The algorithm precises a minimum weight value as parameter (from here on out referred simply as cut weight). Strings with a weight value that is smaller than the cut weight are ignored.
- The aim is to find the substrings shared by two strings which weight is greater than or equal to the cut weight.

- The weight of a target substring might be different in each string.
- A target substring might appear more than once in one of the strings.
- A target substring must not be a substring of another matching substring in at least on of the original strings.

For each target substring complying with the previous conditions, a new embedding feature is created. Its value is the summation of the weights of all the substring appearances in a string. This way, a new embedding space with a finite and small number of features can be built. The number of features for both strings is equal to the number of substrings that comply to the above mentioned conditions. It is possible now to calculate a similarity measure using the inner product between the new feature vectors; this is the so-called *kernel value*. The following is an example that illustrates the proposed kernel function: Let A and B be strings as shown in Figure 3.2. The function $weight_{w \geq n}(A)$ returns the summation of the weights of all the tokens of A which weight is greater than or equal to n. The function $k_{w \geq n}(A, B)$ returns the evaluation of the Kast Spectrum Kernel between A and B. The function $\bar{k}_{w \geq n}(A, B)$ is the normalized version of the former kernel. For $n = 4$ the respective weights are:

$$weight_{w \geq 4}(A) = 64 \tag{1}$$

$$weight_{w \geq 4}(B) = 52 \tag{2}$$

The target in this example are all substrings with weight greater than or equal to 4 (cut weight). According to the kernel definition, three shared substrings are obtained: S_1, S_2 and S_3 (See Figures 3.2, 3.2 and 3.2). The respective weights of each feature in A are calculated with:

$$weight_{w \geq 4}(S_1)_A = 19 \tag{3}$$

$$weight_{w \geq 4}(S_2)_A = 7 + 6 = 13 \tag{4}$$

$$weight_{w \geq 4}(S_3)_A = 6 + 9 = 15 \tag{5}$$

The embedding feature vector for A is:

$$f_{w \geq 4}(A) = \{19, 13, 15\} \tag{6}$$

The respective weights of each feature in B are calculated with:

$$weight_{w \geq 4}(S_1)_B = 17 + 18 = 35 \tag{7}$$

$$weight_{w \geq 4}(S_2)_B = 6 + 5 = 11 \tag{8}$$

$$weight_{w \geq 4}(S_3)_B = 8 + 6 = 14 \tag{9}$$

The embedding feature vector for B is:

$$f_{w \geq 4}(B) = \{35, 11, 14\} \tag{10}$$

The inner product of these two vectors gives us the kernel value

$$k_{w\geq 4}(A,B) =< f_{w\geq 4}(A), f_{w\geq 4}(B) >= 1018 \tag{11}$$

A normalization step will use the weights of each string:

$$\bar{k}_{w\geq 4}(A,B) = \frac{k_{w\geq 4}(A,B)}{\sqrt{k_{w\geq 4}(A,A) * k_{w\geq 4}(B,B)}} = \frac{k_{w\geq 4}(A,B)}{weight_{w\geq 4}(A) * weight_{w\geq 4}(B)} \tag{12}$$

$$\bar{k}_{w\geq 4}(A,B) = \frac{1018}{64 * 52} = \frac{1018}{3328} = 0.3059 \tag{13}$$

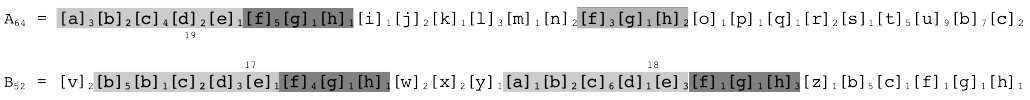


Fig. 3. S_1 is the largest substring found on both examples

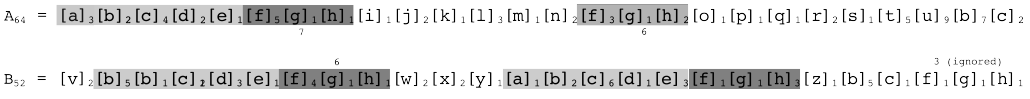


Fig. 4. S_2 appears once as an independent case

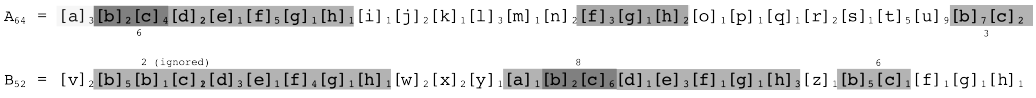


Fig. 5. S_3 appears twice as an independent case

4 Evaluation

4.1 Experiment Configuration

The I/O access patterns were taken from two different parallel I/O benchmarks ([14] and [15]). The patterns were generated from 4 different I/O forms of accessing the storage: (A) were those using Flash I/O, (B) were the ones using

Random POSIX I/O, (C) were those using Normal I/O and (D) the ones using Random Access I/O. For each pattern 4 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. So, from 22 examples we ended up with 110, distributed as follows: (A) 50 examples, (B) 20 examples, (C) 20 examples and (D) 20 examples. Each access pattern was converted to the two proposed string representations: the one that took into account the byte information of the operations and the one that totally ignored it. The proposed *Kast Spectrum Kernel* function was applied to them, as well as the *Blended Spectrum Kernel* proposed in the literature. 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 Kast Spectrum Kernel

The application of the proposed kernel function (*Kast Spectrum Kernel*) over strings that preserved the byte information from the I/O operations, achieved the best results when a small cut weight was used. The fact that small cut weights were sufficient to achieve a meaningful clustering, eased the parametrization of the comparison process. It was remarkable that both learning algorithms clearly separated the same 3 clusters (See Figures 4.2 and 4.2). 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 roughly the same pattern. Also, it is important to notice that there were not misplaced examples on any of the groups.

In the case of the strings that ignored the byte information, such clear separation of clusters was not so easily achieved. For small cut weights only two clusters were identified: Random POSIX I/O (B) was the only group independently separated, while Flash I/O, Normal I/O and Random Access I/O (A-C-D) conformed a second group. In order to obtain the same three clustering groups identified using the other string category, the weight value had to be increased, which made the parametrization more difficult. Notice that, regardless of the string representation, the smaller the cut weight the most expensive the computation became, because the algorithm always started searching from the substrings with the highest weight. According to the clustering analysis results one can infer that the usage of high cut weights is recommended to focus only on finding general categories and lower cut weights to discriminate better among examples. However, a small cut weight is always preferred, as it eases the parametrization.

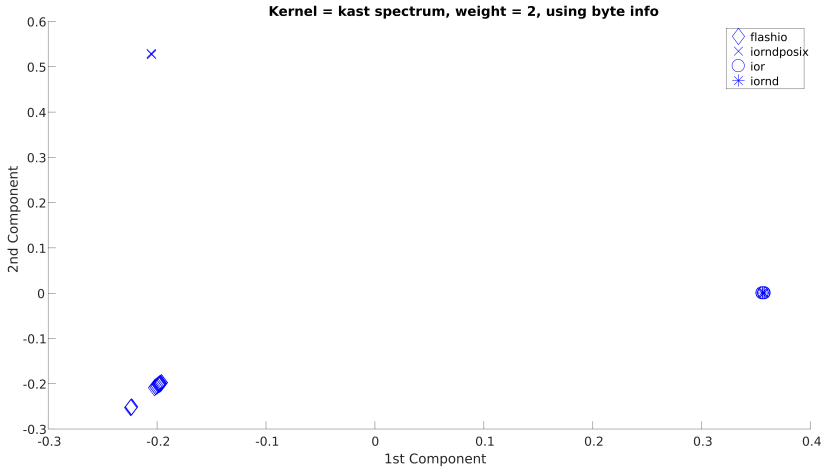


Fig. 6. Kernel PCA for Kast Spectrum Kernel using byte information (cut weight = 2)

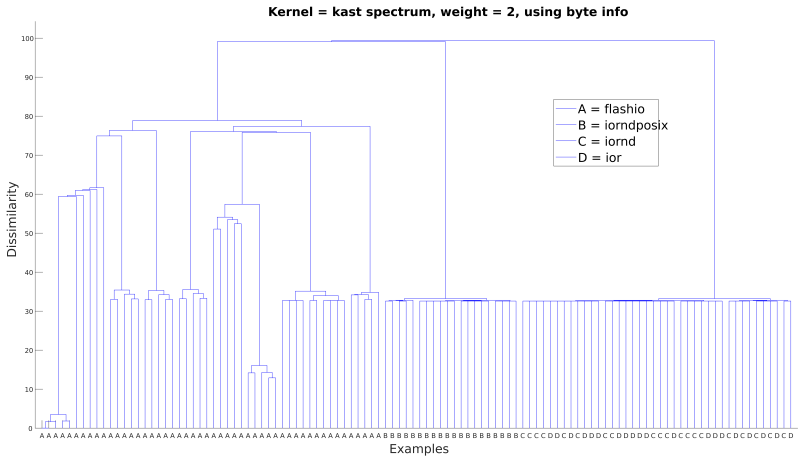


Fig. 7. Hierarchical clustering for Kast Spectrum Kernel using byte information (cut weight = 2)

4.3 Blended Spectrum Kernel

Given the particular form of the string representation we propose, where a group of subsequent tokens can encode more meaningful information than a single one, we discarded the bag-of-characters and the bag-of-words kernels. Experimental evaluation showed also that the k-Spectrum kernel was not successful at finding an acceptable clustering, a task where the Blended Spectrum Kernel had a better performance. However, for strings containing byte information the obtained clusters were not as diverse as those achieved with our solution (See Figures 4.3 and 4.3). In this case only Flash I/O (A) examples were independently separated, while Random POSIX I/O, Normal I/O and Random Access I/O (B-C-D) conformed a single group.

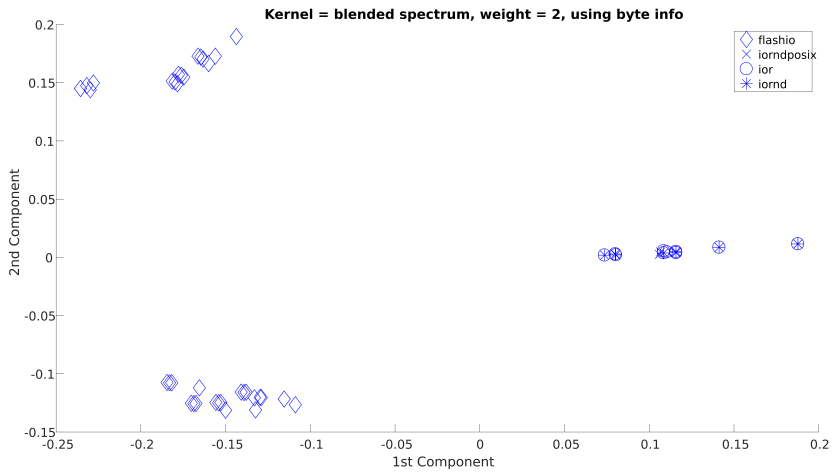


Fig. 8. Kernel PCA for Blended Spectrum Kernel using byte information (cut weight = 2)

For the case of strings lacking the byte information, both clustering analysis results were not satisfactory.

5 Related Work

Kluge [13] 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

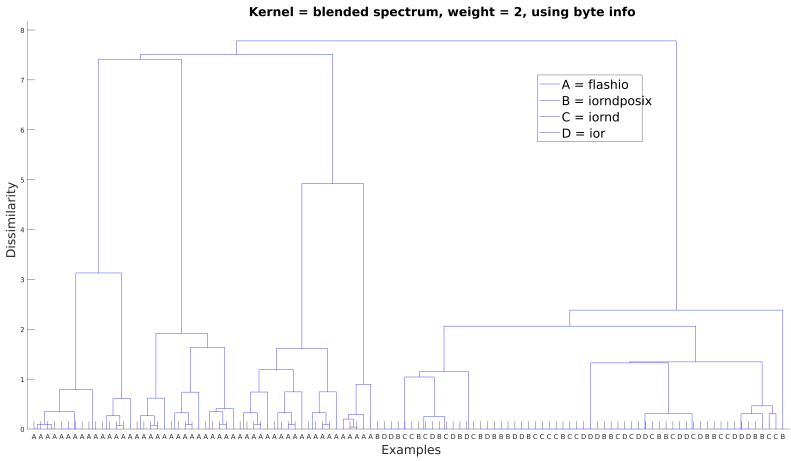


Fig. 9. Hierarchical clustering for Blended Spectrum Kernel using byte information (cut weight = 2)

one. Madhyastha et al. [16] 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 training with previously labeled examples. Behzad et al. [17] 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. [2]. 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 [18] 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 [19] 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 couldn't 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 extract patterns and represent them as a string of tokens. The resulting strings were compared using a novel kernel function proposed by the authors. The Kast Spectrum Kernel emits a similarity matrix between examples that can be later analyzed by a proper algorithm. This kernel was applied to a set of examples

taken from a real parallel application, where 4 distinct patterns were present; Kernel PCA and Hierarchical Clustering showed a consistent formation of 3 groups according to the pattern with no misplaced examples. The best results were obtained when the string representation took into account the byte information of the operations and the cut weight was small. It was observed that the cut weight determined the granularity of the search, while the usage of the byte information permitted the separation between examples of the same cluster. These findings clearly show that both the proposed string representation and the comparison method are suitable to compare I/O access patterns of 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 using the string representation and kernel method here proposed.

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. Kunkel J. M., Simulating parallel programs on application and system level. Computer Science –Research and Development. 28(2), pp. 167–174 (2012)
2. 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)
3. Kung S. Y.: Kernel Methods and Machine Learning. Cambridge University Press. Cambridge, (2014)
4. Shawe–Taylor J., and Cristianini N.: Kernel Methods for Pattern Analysis. Cambridge University Press New York, New York (2004)
5. BakIr G., Hofmann T., Schölkopf B., Smola A. J., Taskar B., and Vishwanathan S.V.N: Predicting Structured Data. The MIT Press (2007)
6. Hastie T., Tibshirani R., and Friedman J.: The Elements of Statistical Learning - Data Mining, Inference, Springer Series in Statistics, Springer-Verlag, New York, (2009)
7. 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)
8. 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)
9. Gärtner, T., Lloyd J. W., and Flach, P. A.: Kernels and Distances for Structured Data. Machine Learning. 57(3), 205–232, (2004)

10. Haussler D.: Convolution Kernels on Discrete Structures. Technical Report. University of California at Santa Cruz, (1999)
11. 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)
12. 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)
13. Kluge M.: Comparison and End-to-End Performance Analysis of Parallel Filesystems. PhD Thesis Dissertation. Technische Universität Dresden. (2011)
14. Loewe W., McLarty T., and Morrone C.: IOR Benchmark. (2012)
15. 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)
16. Madhyastha T. M., and Reed D. A.: Learning to Classify Parallel Input/Output Access Patterns. IEEE Transactions on Parallel and Distributed Systems. 13(8), pp. 802–813. (2002)
17. 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)
18. 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)
19. Koller R., and Rangaswami R.: I/O Deduplication: Utilizing content similarity to improve I/O performance. ACM Transactions on Storage (TOS). 6(3), pp. 13:1–13:26. (2010)