Predicting number of threads using balanced datasets for openMP regions

Jordi Alcaraz[1] · **Ali TehraniJamsaz[2]** · **Akash Dutta[2]** · **Anna Sikora[1]** ·
Ali Jannesari[2] · **Joan Sorribes[1]** · **Eduardo Cesar[1]**

Abstract

Incorporating machine learning into automatic performance analysis and tuning tools is a promising path to tackle the increasing heterogeneity of current HPC applications. However, this introduces the need for generating balanced datasets of parallel applications' executions and for dealing with natural imbalances for optimizing performance parameters. This work proposes a holistic approach that integrates a methodology for building balanced datasets of OpenMP code-region patterns and a way to use such datasets for tuning performance parameters. The methodology uses hardware performance counters to characterize the execution of a given region and correlation analysis to determine whether it covers an unique part of the pattern input space. Nevertheless, a balanced dataset of region patterns may become naturally imbalanced when used for training a model for tuning any specific performance parameter. For this reason, we have explored several methods for dealing with naturally imbalanced datasets for finding the appropriated way of using them for tuning purposes. Experimentation shows that the proposed methodology can be used to build balanced datasets and that such datasets, plus a combination of Random Forest and binary classification, can be used to train a model able to accurately tune the number of threads of OpenMP parallel regions.

✉ Jordi Alcaraz
    jordi.alcaraz@uab.cat

Extended author information available on the last page of the article

Springer

1 Introduction

The increasing heterogeneity and complexity of current HPC systems escalates the difficulty of performance analysis and optimization of parallel applications. This is further compounded as novel problems appear and the number of significant tuning parameters increases. Logically, this fact also affects the premises on which performance analysis and tuning tools are built. We claim that these kind of tools should incorporate new strategies, such as machine learning (ML), to further adapt to the characteristics of current HPC systems.

In this work, we limit the scope of the research to applications parallelized using OpenMP as the mainstream multi-threaded programming model for the HPC community. To be able to incorporate ML techniques into performance analysis and tuning tools, we must be able to first identify the features that allow for the characterization of parallel applications. In this sense, the work we presented in [1] showed that a signature built with the values of a subset of hardware performance counters can be used to characterize a set of OpenMP parallel regions.

However, there is also the problem of building representative and balanced datasets for training purposes which is considered a crucial problem in machine learning and data mining [2] because in the presence of significant imbalances the accuracy cannot be a representative of the true performance of a classifier [3]. In the case of OpenMP regions, this problem can take two different forms:

- with the objective of building a balanced search space of code patterns, how to determine if a given parallel region pattern shall be included in a training set or not.
- with the objective of tuning performance parameters, how to deal with naturally imbalanced datasets.

An illustrative example could help clarify the problem and why it can take two different forms. Suppose a bank wants to use ML to train a model to assess the default risk of its customers. First, the dataset must include instances of all potential customers, that is, it must be **representative** of all potential customers classes. Second, the individuals included in the dataset must be uniformly distributed among these classes, that is, the dataset must be **balanced**. In this way, biases due to over/underrepresented classes or the fact of ignoring some classes could be avoided. However, regardless of the characteristics of each class of clients, the number of clients who will meet their obligations is significantly greater than the number who will not. Consequently, from the point of view of default risk, the set includes two highly imbalanced classes (it is **naturally imbalanced**) and a naive model trained with such a dataset will tend to predict that customers will always pay because this result will maximize the accuracy of the model.

Something similar happens in the case of executions of OpenMP parallel regions, the dataset must be balanced and representative to avoid bias in the models, but, in terms of performance parameters such as the number of threads, its distribution would be highly uneven because a few performance parameter values usually lead to the best performance for many executions.

For tackling the problem of creating representative and balanced datasets we presented in [4] an approach for systematically deciding whether a given OpenMP parallel region pattern covers a unique portion of the search space that is not currently considered covered by in a training set. The search space is the N-dimensional space defined by the valid combinations of N hardware performance counters values. In this space a pattern is an OpenMP fragment of code that abstracts different regions with a similar behaviour. However, a training set that is balanced and representative of a space of OpenMP patterns' executions may be naturally imbalanced when used for optimizing the value of a performance parameter. Consequently, we have focused our efforts on exploring the main approaches for dealing with naturally imbalanced datasets for tackling the natural imbalance of a training set used for tuning the number of threads of a region.

Therefore, the objective of this paper is to present a holistic perspective for building representative and balanced datasets of OpenMP patterns and tackling their natural imbalance when used for tuning purposes. To fulfill this objective, we summarize the contribution made in [4] introducing a refinement for reducing the number of executions of the patterns, and present the study that has been done to determine which are the best approaches for dealing with naturally imbalanced datasets for tuning performance parameters.

Consequently, the main contribution of this work is to introduce a methodology that is able to: 1) build a balanced and representative training set of OpenMP parallel regions and 2) overcome its natural imbalance to train a model able to predict the best value of a tuning parameter (as a use case, the number of threads).

This work is organized as follows. Section 3 summarizes the methodology for determining if a given parallel region pattern shall be included in the dataset. Section 4 shows how the proposed methodology can be used to build and validate a dataset using a set of kernels extracted from different benchmarks. Section 5 describes the study of the different methods for dealing with natural imbalanced datasets for the particular case of predicting the best number of threads for an OpenMP region. Related works are discussed in Sect. 2. Finally, Sect. 6 concludes this paper and discusses potential future work.

2 Related work

Similar works appear in the literature which focus on classification of parallel regions by leveraging analytical models or machine learning to identify parameters such as the number of threads or scheduling.

DiscoPoP [5] is a tool designed to locate opportunities for parallelism in sequential programs and give suggestions to the user as to their potential implementations. DiscoPoP identifies small regions of code called computational units (CU), which follow the read-compute-write pattern, and arranges them in a data dependency graph. The dependencies identified by DiscoPoP are used in order to find 4 types of patterns [6]. For each of them an OpenMP construct is proposed for producing the parallelized version of the input program. In [7] dynamic features generated by DiscoPoP are used in order to train classifiers to classify loops in a sequential program as parallelizable or

not. The work in [8] presents a similar approach using program dependence graphs and other features to train an ML model used to predict the probability of a code regions being parallelizable. In [9], profiling data is used to extract information about a region and SVM are used to decide whether a candidate loop should be parallelized or not. These approaches use ML techniques for discovering parallelizable regions, while our proposal uses these techniques for automatically tuning performance parameters associated to each parallel region.

Kernel Tuning Toolkit (KTT) is an autotuning tool for OpenCL and CUDA kernels [10]. KTT uses an automatic search of the tuning space to determine a configuration for one or more kernels with shared tuning parameters. In [11] performance is considered a function of performance counters and hints that the relationship between tuning parameters and performance counters is portable for different GPUs and problem inputs. Decision Trees are used to predict performance counters for unknown configurations. A benchmark redundancy reduction approach is presented in [12]. The main idea is to extract non-overlapping sections of code from benchmarks, characterize them using a set of static and dynamic features, and use clustering to select the subset of representative sections for accelerating system selection.

The approach in [13] focuses on using active-learning instead of empirical evaluation for performance tuning. It uses a particular implementation of regression trees called dynamic trees in order to find an appropriate selection of parameter values inside a defined parameter space. In [14], a compiler based approach is taken to predict the number of threads and scheduling policy of OpenMP applications. This approach uses ANN to predict scalability and Support Vector Machines (SVMs) to select the scheduling policy.

APARF [15] is an adaptative runtime framework to enhance peformance of OpenMP task-based programs. It makes use of PAPI to obtain performance events and trains an ANN to find which scheduling scheme should be used to obtain the optimal performance in unseen programs.

All these works propose using ML techniques for tuning a set of performance parameters using data collected during the execution of the application. This indicates that, like us, several other groups are biding for these techniques in order to overcome the difficulties introduced by the growing complexity of current HPC hardware and software. However, to the best of our knowledge, none of them tackle the problem of building and using balanced and representative training sets as we propose in this paper, which could jeopardize the results obtained.

3 Generating balanced dataset of OpenMP patterns

This section summarizes the methodology presented in [4] to determine whether a given OpenMP parallel region pattern covers a portion of the feature space that is not represented by the training set so far. Finding an effective solution to this issue is significant in ensuring a balanced and representative dataset. Intuitively, including one or more patterns which are nearly identical will not provide any further information and are a source of imbalance which reduces the quality of the dataset.
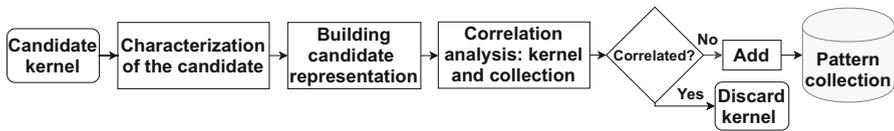
Fig. 1 Methodology for determining whether a pattern should be included or not in the training set

This methodology assumes that the set of hardware counters necessary to characterize the execution of a code region has been previously determined. We have used the results obtained in [1], which shows that, for the architecture used in this work, 18 hardware counters (of the 58 generic counters available) are enough to characterize the execution of an OpenMP parallel region. This set includes counters related to branch prediction, use of the cache hierarchy, and the type and number of instructions executed.

Figure 1 shows the general scheme of the proposed methodology. The *pattern collection* represents the current training dataset, which we assume to be balanced. We consider that if the signature[1] of a *candidate kernel* is not highly correlated with any pattern in the *pattern collection*, it covers a new part of the input space and should become a new pattern in the collection.

In the case that we locate an OpenMP kernel which appears to be an attractive candidate for inclusion in the dataset, we use the following steps to verify whether or not it covers a new portion of the input space:

– **Characterization of the candidate kernel**. The kernel is executed to obtain its signature for each combination of number of threads and affinity.
– **Building candidate kernel representation**. After the kernel's data is available, the signatures are joined creating a representation of the *candidate kernel* as a vector that will be used in the correlation analysis.
– **Correlation analysis**. Correlation analysis is performed between the new vector and the vectors of each pattern already in the *pattern collection* to determine the extent of their similarity. If the candidate kernel is highly correlated with at least one pattern in the collection, it is discarded. Otherwise, it is considered to cover a new portion of the input space and it is added to the collection as a new pattern.

3.1 Characterizing and building the representation of the candidate kernel

The first step for assessing a candidate kernel consists of generating its signatures. In order to obtain the kernel's signature for the OpenMP parallel region it must be executed multiple times (for ensuring statistical significance) for all possible combination of the *number of threads* and *thread affinity* parameters.

In addition, with the objective of increasing the accuracy of the kernel's signatures, multiple executions are required to obtain all the counters' values, where each execution measures one set of counters. Consequently, the total number of executions of the candidate kernel can be computed using Eq. (1).

$$n\_executions = n\_times * counter\_sets * thread\_affinities * n\_cores \quad (1)$$

---

[1] a signature is defined as values of a representative set of hardware counters [1]

All the information gathered in these executions is used to build a vector of signatures ordered by the number of threads, affinity and repetition number.

3.2 Correlation analysis

The last step of the methodology consists of performing a correlation analysis between the candidate kernel representation (vector of signatures) and the representation of all patterns in the current *pattern collection*.

Spearman's rank correlation and Kendall's tau [16] are applied to establish whether the candidate kernel covers a new region of the input space or not. If both methods indicate that the candidate kernel is not highly correlated to any patterns in the current collection and it logically represents a different region, the candidate kernel will become a pattern and be added to the collection.

The reason for using these two correlations is that Spearman's rank correlation is based on deviation between two series of data and is more sensitive to data errors and discrepancies, while Kendall's tau is based on concordance (and discordance) of data pairs and is effective for detecting trends. In almost all cases, both of them will lead to the same inferences and so the verification of this serves to increase the robustness of our method.

Consequently, if the candidate kernel is highly correlated with at least one pattern in the collection, i.e., it is at least correlated to the signature corresponding to any problem size of one patterns already in the dataset with a Spearman's $r > 0.9$ and Kendall's $tau > 0.8$, the kernel is discarded as it does not cover a new portion of the input space. Otherwise, the candidate kernel is considered to cover a new portion of the input space and it is added to the collection as a new pattern.

For doing so, the kernel must be executed for all the problem sizes included in the *pattern collection* to obtain the corresponding signatures. Initially, we had selected 43 problem sizes ranging from small (few thousands elements in each vector) to huge problems sizes (a few billion elements) with the objective of stressing every level of the memory hierarchy. Later, we wondered if it was possible to reduce the number of problem sizes without reducing the quality of the dataset and, therefore, reduce the size of the dataset and the time to compute all the signatures for a new pattern (Eq. 1 by 43 sizes).

Therefore, we have defined a systematic way for determining the problem sizes. The general idea is that small problem sizes are defined using the size of the different cache level in one socket, and bigger problem sizes are proportional to the size of the last level cache (LLC) and the number of sockets.

Specifically, we use the knowledge about the system memory hierarchy and proceed in the following way (the process is illustrated using the XEON E5645 processor shown in Fig. 2):

- To stress the private cache levels (L1 and L2 in example) `number of private levels x number of cores` problem sizes are defined (12 in the example), starting with the size of one lowest level cache (32KB in the example) and ending with the accumulated size of all private caches (1.5MB in the example as shown in Fig. 2b). So, in the example, the resulting small problems sizes would be (in
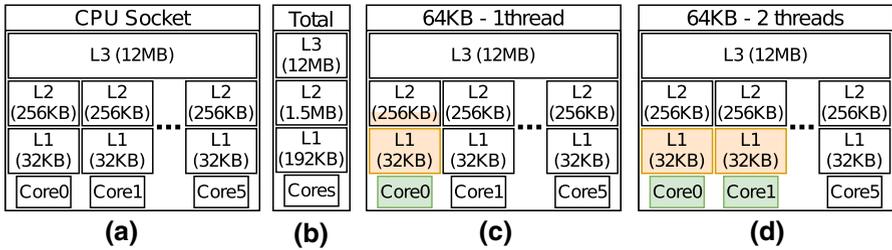
Fig. 2 Memory hierarchy of XEON E5645's socket: **a** size of each memory level; **b** total memory per level; 64KB problem size with one thread (**c**) and two threads (**d**)

KB): 32, 64, 96, 128, 160, 192, 256, 512, 768, 1024, 1280 and 1536. It is worth noticing that for certain combination of number of threads there will be access to the L3, but, in general, we are keeping most of the memory accesses in the private cache levels.

– To stress the shared cache levels (L3 in the example) `six` problem sizes are defined; the first slightly bigger that the accumulated size of all private caches (2MB in the example) and the last slightly smaller that the shared cache size (11.5MB in the example). In this case, we are sure that most memory accesses are done within the cache hierarchy, specially in the L3.

– Using RAM. The biggest problem sizes are defined to force a significant number of accesses to the main memory. In this case, we define `number of sockets + 1` problem sizes, starting with a size slightly bigger that the LLC size (12.5MB in the example), gradually increasing the size of the problem in the same amount for each socket (second problem of 25MB in the example), and generating a last problem size (the biggest case) incremented by a 50% (37.5MB in the example).

This approach achieves two significant objectives:

1. The number of problem sizes (21 for the presented example) is tailored to the complexity of the underlying hardware, i.e., the dataset will also be balanced with respect to the memory hierarchy because the number of cases will be adequate to reflect its characteristics.
2. We have a more precise control of which executions are stressing each memory level. This opens the possibility of future refinements for dataset design and its use to train ML models.

4 Validation of the methodology

The proposed methodology has been applied to incrementally construct a dataset of OpenMP parallel patterns and an artificial Neural Network (ANN), described below, has been leveraged to validate the set and demonstrate the importance of using a balanced dataset. Figure 3 summarizes the workflow followed in this validation:

– Phase 1: Building the *pattern collection* applying the steps described in Sect. 3.

Fig. 3 Summary of the experimentation workflow

Table 1 Artificial Neural Network Architecture

Layer	Neurons	Inputs	Activation	Weight Constraint	Dropout
Input	N/A	18	N/A	N/A	0%
Hidden 1	18	18*18	SELU	Clip [− 10.0, 10.0]	10%
Hidden 2	16	18*16	SELU	Clip [− 10.0, 10.0]	10%
Output	8	16*8	Softmax	Clip [− 10.0, 10.0]	0%

– Phase 2: Validating the collection by showing that an ANN trained for classifying code regions using a balanced dataset is highly accurate, even for patterns that have not been considered in the creation of the dataset. It is worth noticing that [4] also shows that the accuracy of the trained model is significantly higher when using a balanced dataset than when using an imbalanced one.

The candidate kernels for building the training set have been extracted from two different well-known benchmarks:

– STREAM (Sustainable Memory Bandwidth in High Performance Computers) [17] is a synthetic benchmark composed of simple vector kernels to measure sustainable memory bandwidth.
– POLYBENCH [18, 19] is a collection of benchmarks with multiple kernels. Its version 4 includes 23 different benchmarks divided in different categories (datamining, linear algebra, medley and stencils).

The ANN used for this experimentation and hardware is a Fully-connected, Feedforward Neural Network and its architecture is shown in Table 1.

Scaled Exponential Linear Units (SELU) activations were selected for the hidden layers of the network, which provide several benefits: self-normalizing, cannot die as Rectified Linear Units do, and do not produce vanishing or exploding gradients [20]. The output layer of the network uses a Softmax function paired with Categorical Cross-Entropy, allowing the network to perform classification for multiple classes. Therefore, the network outputs a vector of probabilities for a given instance in the dataset belonging to each pattern.

All the experiments were conducted on a DELL T7500 with two XEON E5645 processors. Each processor, depicted in Fig. 2, features six multi-threaded cores, access to 96GB of shared RAM and three levels of cache (32KB L1 and 256KB L2 for each core, and 12MB L3 shared between all the cores). In addition, [1] shows that OpenMP

Table 2 Spearman(S) and Kendall(K) maximum correlation coefficients for STREAM

	Copy		Triad		Add		Scale	
	S	K	S	K	S	K	S	K
Copy			0.82	0.78	0.86	0.82	0.85	0.83
Triad	0.82	0.78			0.99	0.95	0.99	0.93

parallel regions in this platform can be characterized by the values of 18 hardware counters.

To characterize a candidate kernel on this platform all the combinations (9000 according to Eq. (1)) of the following parameters must be executed:

– *Number of threads*: from 1 to 12 threads.
– *Thread affinity*: close and spread.
– *Number of executions*: 75 to attain statistical significance.
– *Number of counter sets*: 5 to cover the 18 counters.

To fully characterize a pattern on this platform, all these combinations are repeated using 21 different *problem sizes* to stress all the memory levels (as explained in Sect. 3.2). MATE [21, 22] is used to acquire the hardware performance counters values for each of these executions and to compute the signatures for each kernel.

4.1 Building the pattern collection

The first group of candidate kernels is extracted from the STREAM benchmark. It contains four simple OpenMP kernels designed to represent the behavior of vector style applications:

– *Copy*. A simple copy between two vectors: $c[i] = a[i]$.
– *Add*. Addition of two vectors: $c[i] = a[i] + b[i]$.
– *Scale*. Scalar multiplication: $c[i] = scalar * a[i]$.
– *Triad*. Mix of Add and Scale: $c[i] = scalar * a[i] + b[i]$.

As the *pattern collection* is initially empty, it is initialized using the Copy kernel and then the methodology is applied for the remaining candidates.

Once the *pattern collection* has been initialized with all the Copy signatures, and one significant signature has been computed for Add, Scale and Triad, we perform correlation analysis. Table 2 shows that the values of Spearman's rank and Kendall's tau between the three candidate kernels (Add, Scale and Triad) and the pattern in the collection (Copy) are below the conditions (r> 0.9 and tau> 0.8) established in the methodology for considering them to be covering the same region of the input space. However, Table 2 also shows that there is a very strong correlation between Add, Scale and Triad, which is clearly above the threshold.

As a result, only one of the candidates has been included in the *pattern collection*. The chosen pattern is Triad because it is the result of the composition between Add and Scale, i.e., it is logically the most general one. Consequently, after applying the
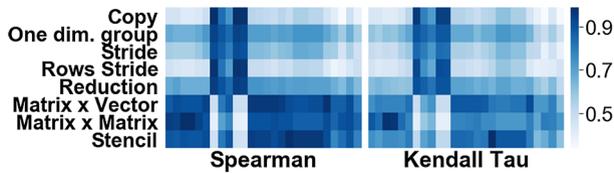
Fig. 4 Spearman rank and Kendall's tau correlation between POLYBENCH discarded kernels and the *pattern collection*

methodology to the kernels extracted from STREAM, the *pattern collection* contained the following patterns:

- *Copy*. Pattern abstracting accesses (reads and/or writes) to consecutive memory positions.
- *One dimensional group*. Pattern abstracting operations involving only one-dimensional vectors.

Next, we extend the *pattern collection* using POLYBENCH, from which we extracted 29 new candidate kernels. After executing all the parallel kernels of POLYBENCH and generating their candidate kernel representations (signature), we incrementally detected new patterns and added them to the *pattern collection*. These new detected patterns are described as follows:

- *Reduction*. Pattern abstracting reduction operations, such as adding all the elements of a vector: $total = \sum c[i]$.
- *Stride*. Pattern abstracting accessing elements with a stride: $c[stride \cdot i] = a[stride \cdot i]$.
- *Rows stride*. Pattern abstracting operations involving column-wise traversal of a matrix: $c[i \cdot N][j] = a[i \cdot N][j]$.
- $Matrix \times Vector$. Pattern abstracting different matrix-vector operations, such as the matrix-vector multiplication: $A = B \times v$.
- $Matrix \times Matrix$. Pattern abstracting different matrix-matrix operations, such as the matrix-matrix multiplication: $C = A \times B$.
- *Stencil*. Pattern abstracting stencil operations, such as: $A[i][j] = A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1]$

Figure 4 shows the correlation of the kernels extracted from POLYBENCH and the 8 detected patterns. It can be seen that all these kernels are highly correlated to at least one of the patterns in the collection.

Summarizing, the dataset obtained from the STREAM and POLYBENCH benchmarks is composed of the following patterns: *Copy*, *One dimensional group*, *Stride*, *Rows stride*, *Reduction*, $Matrix \times Vector$, $Matrix \times Matrix$, *Stencil*.

4.2 Validating the pattern collection

In order to validate the collection obtained in the previous subsection, which also validates the proposed methodology, we have used this collection to train the ANN for producing a pattern classification model.

Table 3 Accuracy of the ANN when using the kernels that do not represent any pattern in the collection

	Number of kernels	Accuracy
Copy	3	93%
1D group	3	90%
Stride	3	91%
Rows stride	0	–
Reduction	0	–
Matrix × Vector	5	90%
Matrix × Matrix	5	87%
Stencil	12	99%

The pattern dataset is composed of 302400 signatures representing the 8 detected patterns. The dataset is divided randomly into a training set composed of the 80% signatures and a test set composed of 20% of the signatures. The network has been trained for 24 epochs using batches with 100 signatures and obtaining a final loss of 0.0301 and accuracy of 98.93%.

Next, a validation set has been built using the signatures of the kernels considered, but discarded in the previous section, and several new kernels extracted from the NAS Parallel Benchmarks (NPB) [23]. Specifically, we used the following ten OpenMP parallel kernels extracted from the NPB:

– *Add_BT* and *rhs_norm_BT*. These kernels correspond to the *add* and *rhs_norm* BT's functions, respectively.
– *normztox_CG*, *norm_temps_CG*, *rhorr_CG*, *z_alpha_p_CG*, *pr_beta_p_CG*, and *qAp_CG*. They are regions that have been extracted from different CG's functions.
– *l2norm_LU*. It corresponds to the *l2norm* LU's function.
– *ssor_LU*. It is a region extracted from the LU's *ssor* function.

Table 3 shows the very high accuracy of the trained classification model on the signatures of the kernels extracted from the STREAM and POLYBENCH benchmarks that do not have been chosen for building any of the pattern representations (discarded kernels).

Table 4 shows the results produced by the pattern classification model for the 10 kernels extracted from the NPB. In this case, we are proceeding the other way around because there was no previous correlation analysis for telling us to which pattern is correlated each of these kernels. Consequently, to validate the classification offered by the model, we have computed the correlation coefficients of each kernel signature to the patterns in the dataset. Figure 5 shows both correlation coefficients (Spearman's rank and Kendall's tau) between the NPB candidate kernels and the patterns in the dataset. It is clearly observed that the plotted results align with the classification given by the model. In addition, these kernels have been manually analyzed to confirm that they correspond to the detected patterns. These results demonstrates that the ANN can also be used to detect candidate patterns not included yet in the dataset.

Table 4 Classification of the kernels extracted from the NPB using the ANN

NAS kernel	Predicted Pattern
Add_BT	One dimensional group (97%)
l2norm_LU	Reduction (88%)
norm_temps_CG	Reduction (100%)
normztox_CG	One dimensional group (100%)
pr_beta_p_CG	One dimensional group (99%)
qAp_CG	Reduction (94%)
rhorr_CG	Reduction (100%)
rhs_norm_BT	Reduction (88%)
ssor_LU	One dimensional group (98%)
z_alpha_p_CG	One dimensional group (84%)
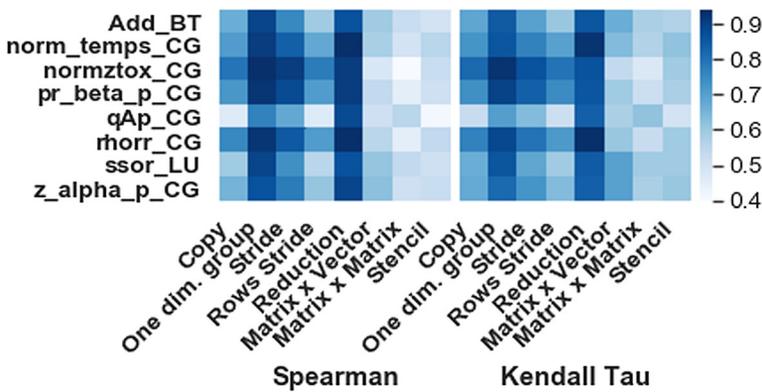
Fig. 5 Correlation coefficients between NPB kernels and the *pattern collection*

5 Naturally Imbalanced Datasets: Predicting the ideal number of threads

Using the methodology presented in Sect. 3 it is possible to create datasets where no parallel region patterns and problem sizes are underrepresented. However, when it comes to tuning a performance parameter, the dataset may become naturally imbalanced because it is natural for some values of the parameter to be the best for a significant number of cases.

In this section, we present the results obtained after exploring several techniques when dealing with this kind of dataset for tuning the number of threads of an OpenMP region using the dataset built in Sect. 4.1.

First, we show how the ideal number of threads is determined and why the considered dataset becomes naturally imbalanced when used for predicting the value of this performance parameter. Next, we present the results obtained when using this dataset for training an ANN similar to the one used in Sect. 4.2, and also the ones generated using a decision tree with the default parameters of *scikit-learn*. In the case of cost-sensitive learning, weights are applied to classes with the parameter: *class_weight=balanced*. In this way, we highlight that the problem lies in the training
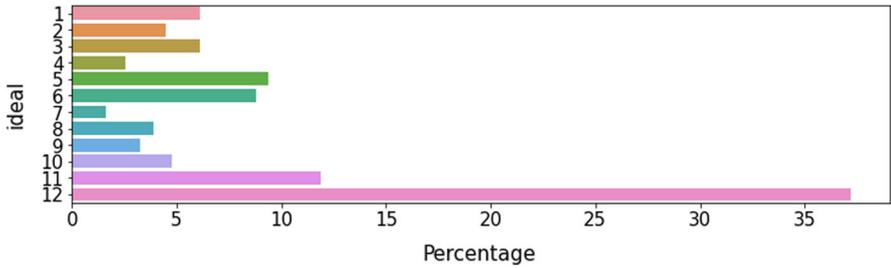
Fig. 6 Ideal number of threads, according to $Pi(X)$, for all the executions stored in the dataset

set because results are unsatisfactory for both methods. In addition, we established a base case for comparison. Finally, we show and discuss the results obtained for each of the explored methods and point out the one that has produced the best results so far.

5.1 Ideal number of threads and consequences of dataset natural imbalance

For determining the ideal number of threads for a given execution of a kernel, we use the performance index defined in [24]. This index, computed using Eq. 2, relates the execution time of a kernel using a certain number of threads ($T_t(X)$) with the efficiency obtained for that number of threads ($E(X)$). The value that minimizes ($Pi(X)$) is the number of threads (X) that optimizes the performance (time) without wasting resources (cores).

$$Pi(X) = \frac{T_t(X)}{E(X)} = \frac{X \cdot T_t(X)^2}{T_t(1)} \qquad (2)$$

Figure 6 shows the results of optimizing $Pi(X)$ for all the executions stored in the dataset. It can be seen that 7 is the best number of threads only in 1.6% cases, while 12 is the best in 37%. There is a clear imbalance that cannot be fixed by eliminating elements from the dataset without loosing too much information. Nevertheless, we have used the dataset as it is for training models with different ML techniques (SVM, logistic regression, KNN, ANN and Decision Trees DT). We have kept the models produced by the ANN and the DT, because they are significantly more accurate than the rest.

The training of these techniques has been performed using the patterns in the collection and the resulting models have been tested using the same patterns plus a set of non perfectly correlated kernels. We believe that this experiment is more realistic than using the leave one out cross-validation without checking if there are still similar kernels in the training set. Summarizing:

– **Patterns in the training set** (8): *copy*, *reduction*, *rows*, *stride*, *1D-Group*, $Mat \times Mat$, $Mat \times Vect$ and *stencil*, which correspond to the groups determined in Section 4.1.
– **Patterns in the test set** (16): the ones in the training set plus *2D-Stencil*, *1D-Stencil*, *scale*, *add*, *stride16*, *stride2*, *stride4* and $Mat \times Mat Opt$.

Considering Eq. 1 and the number of problem sizes (21), the number of signatures stored in the dataset for each pattern is 37800, that is, 302400 signatures in the training set and 604800 signatures in the test set. Consequently, it is important to highlight that the objective of this exploration is not to determine the performance improvement for each case, but the accuracy of the trained models in determining the number of threads that maximizes, according to Eq. 2, the performance of an OpenMP code region.

Figure 7a and 7b show the results for a simple multi-layer ANN and a Decision Tree. Results are displayed with stacked bars showing: the number of times the prediction successfully finds the ideal thread number (*First*); the number of times the predicted number is the second best value using the performance index (*Second*); and the number of times the prediction underestimates the number of threads which at least does not waste resources (*Higher*).

In this case the model created by DT obtains better results in almost all cases. When tested with cases used in the training, DT shows almost 100%, while ANN gets around 80%. In the other cases, except for the stride kernels, DT gets at least 60% accuracy between *First* and *Second*, while ANN shows poorer results. The prediction for the stride kernels is consistently underestimating the number of threads, which hints that this case should be further analyzed.

5.2 Exploring methods for dealing with naturally imbalanced datasets

Once we realized that the imbalance in the dataset is tampering the trained models, we focused on exploring the main ways to deal with it [25]:

– **Data methods** are re-sample strategies which may consist of under/over-sampling using the available data or generating new synthetic data. Oversampling can be achieved with random replication (suffers from overfitting [26]) or with the generation of synthetic data with different heuristics, such as SMOTE [27].
– In **algorithmic methods**, existing machine learning techniques are modified to prevent underrepresented classes from being ignored. Among these approaches we have cost-sensitive learning [28].
– **Ensemble methods** train multiple learning models to obtain a prediction, each model can be obtained using the same or different machine learning techniques. For obtaining the overall results, the methods runs all the trained models and applies the decision criterion, such as the average or weighted average of the answers, or the most frequent answer. An example of a well known ensemble method is Random Forest, which is an ensemble of multiple decision trees, generated using different data from the same dataset [29].

First, we tested the re-sampling strategies: random over-sampling and multiple SMOTE implementations. We quickly realized that they were not producing the expected results. For example, Figure 8 shows that using SMOTE the accuracy in the test cases decreases for all the kernels.

Next, we explored the cost-sensitive algorithmic approach to counter the imbalance in the dataset applying inversely proportional costs (error) to the frequency of each label, so the errors are proportionally bigger in the less represented labels than the errors for the most represented labels. Figures 7c and 7d show that both models have
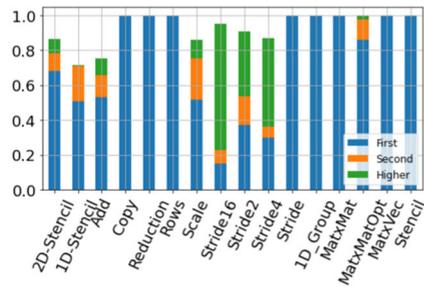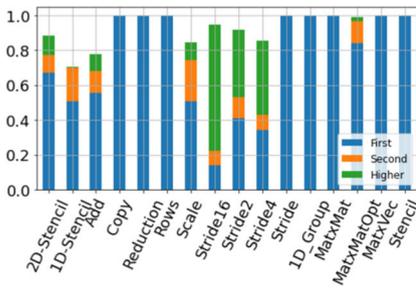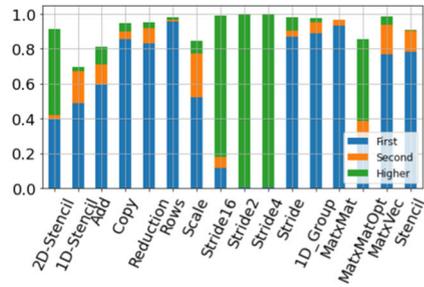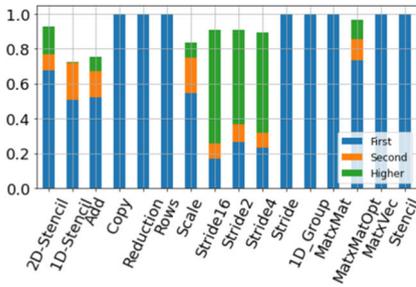
(a) Decision Tree

(b) Artificial Neural Network

(c) Decision Tree with cost

(d) Artificial Neural Network with cost

(e) Random Forest

(f) Ensemble Artificial Neural Network

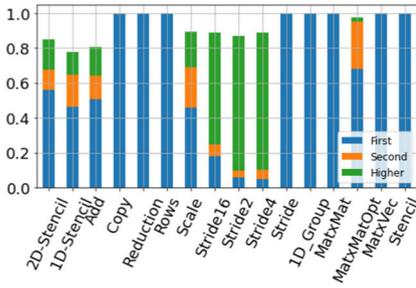**(g)** Random Forest binary classifier

(h) Random Forest binary classifier with cost
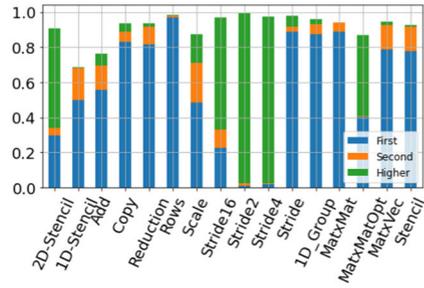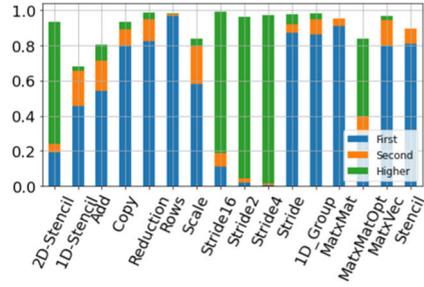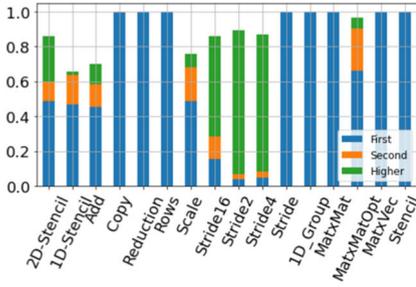
Fig. 7 Predicting the number of threads using machine learning and multiple balancing approaches

(a) Decision Tree   **(b)** Artificial Neural Network
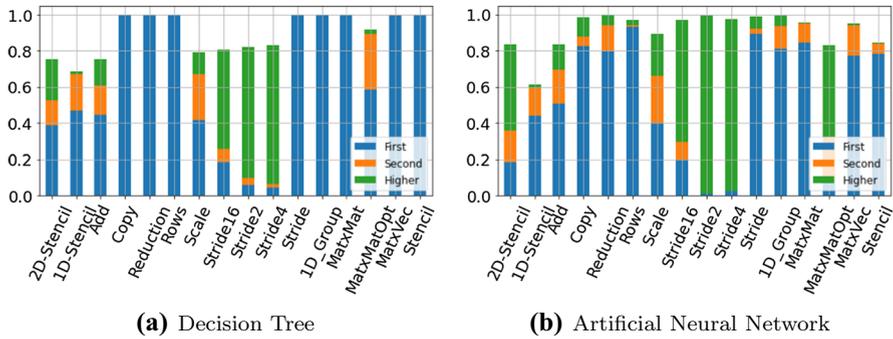
Fig. 8 Predicting the number of threads balancing using SMOTE

improved the accuracy, especially the one produced by an ANN. However, the model produced by the DT is still significantly more accurate.

These results are encouraging, but we thought that there was still room for improvement, so ensemble learning methods has been also explored. In the case of DT, Random Forest is the natural ensemble model. The random forest was implemented using *scikit-learn* with the default configuration which uses 100 estimators, where each estimator uses a random subset of features. In addition, an ensemble of multiple ANNs with the same architecture has been used, in this case, the number of estimators is equal to the number of features (19, 18 counters + thread affinity). In both cases, each model is trained with a subset of the data, generating different models which depend on the randomly selected data in each case.

Figure 7e shows that ensemble of ANNs did not improve the accuracy of the prediction, although the training time of the ensemble is 20X higher that the one of a single ANN (approximately 10 hours). However, Fig. 7f shows that the accuracy of the ensemble model produced by the Random Forest has significantly improved especially for the stride kernels. In these cases, although the accuracy remains low, it has more than doubled.

Finally, we tested the one-against-all binary classification [30] approach with and without cost-sensitive learning. The problem is divided into 12 different models, each class being a thread configuration, where each individual model is trained to identify only one class. This test has only been done with Random Forest as it has shown better accuracy than the ANN.

The results (shown in Fig. 7g) with binary classification improve the ones obtained with the previous Random Forest. Two strides cases achieve more than 40% accuracy for the two best cases in the performance index, and matrix multiplication reaches a 96%. For the remaining classes, the accuracy is above 70%. On the contrary, applying cost-sensitive and binary classification all-together does not increase the accuracy. This is because with binary classification all of available information about the other classes is discarded, consequently this affects the calculated loss value.

This exploration of methods when dealing with naturally imbalance datasets is not yet exhausted, e.g. we are still analyzing the use of GANS as a re-sampling method because of the precision of this kind of networks to capture the distribution of the

elements of a dataset [31]. However, the results obtained so far show that advanced methods, in particular the combination of Random Forest with binary classification, for dealing with naturally imbalanced datasets reach an accuracy level that hints the robustness of our initial hypothesis about the advantages of incorporating ML into performance analysis and tuning tools.

6 Conclusions and future Work

We have developed an holistic perspective for building balanced and representative datasets, tackling their natural imbalance when used for training a ML approach for tuning performance parameters. For building a dataset, the methodology is focused on determining whether a given OpenMP parallel region pattern covers a portion of the feature space that is not represented in the set so far. In this proposal, an OpenMP pattern is represented by the signatures of a representative kernel, calculated for various problem sizes, and correlation is used to decide if a given kernel could be representative of a pattern covering a new portion of the feature space.

The use of the proposed methodology has been illustrated through its application to the set of kernels extracted from well-known benchmarks (STREAM and POLY-BENCH) for a particular architecture. This process has allowed for the construction of a preliminary dataset, which has been used for training a model for optimizing the number of threads executing a parallel region.

For training this model, we have had to deal with the natural imbalance of a dataset when it comes to tuning performance parameters and have found, after an extensive exploration, that a combination of the ensemble method Random Forest and binary classification produces the highest accuracy.

The results obtained show that the proposed methodology allows for systematically generating balanced datasets, that using a balanced datasets has a significant impact on the classification accuracy, and that the ML method for tuning performance parameters must be carefully chosen to avoid the effects of the natural imbalance appearing in this case.

Our next step will be to extend the methodology for using datasets to train robust ML methods for tuning more performance parameters. Upon reaching this goal, it is conceivable that these methods could be applied to automatically tune the combination of key performance parameters (number of threads, affinity, or scheduling policy) that will be used in a given region. In conclusion, the holistic perspective described in this work is a significant step towards an autonomous system for addressing performance issues for OpenMP regions.

References

1. Alcaraz J, Sikora A, César E (2019) Hardware counters' space reduction for code region characterization, in Euro-Par 2019, ser. Lecture Notes in Computer Science, R. Yahyapour, Ed., vol. 11725. Springer, 2019, pp 74–86
2. Nguyen GH, Bouzerdoum A, Phung SL (2009) Learning pattern classification tasks with imbalanced data sets, in Pattern Recognition, P.-Y. Yin, Ed. Rijeka: IntechOpen, ch. 10
3. Nath A, Subbiah K (2018) The role of pertinently diversified and balanced training as well as testing data sets in achieving the true performance of classifiers in predicting the antifreeze proteins. Neurocomputing 272:294–305
4. Alcaraz J, Sleder S, TehraniJamsaz A, Sikora A, Jannesari A, Sorribes J, Cesar E (2021) Building representative and balanced datasets of openmp parallel regions, In: 2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pp 67–74
5. Li Z, Jannesari A, Wolf F (2013) Discovery of potential parallelism in sequential programs, In: 42nd international conference on parallel processing, pp 1004–1013
6. Norouzi M, Wolf F, Jannesari A (2019) Automatic construct selection and variable classification in openmp. Proc ICS 2019:330–341
7. Fried D, Li Z, Jannesari A, Wolf F (2013) Predicting parallelization of sequential programs using supervised learning, In: 2013 12th international conference on machine learning and applications, vol. 2. IEEE, pp 72–77
8. Maramzin A, Vasiladiotis C, Lozano R, Cole M, Franke B (2019) It looks like you're writing a parallel loop: a machine learning based parallelization assistant, In: Proceedings of the 6th ACM SIGPLAN International Workshop on AI-SEPS, 2019, pp 1–10
9. Tournavitis G, Wang Z, Franke B, O'Boyle MF (2009) Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. ACM Sigplan notices 44(6):177–187
10. Filipovič J, Petrovič F, Benkner S (2017) Autotuning of opencl kernels with global optimizations, In: Proceedings of the 1st Workshop on AutotuniNg and ADaptivity AppRoaches for Energy Efficient HPC Systems, ser. ANDARE' 17, NY, USA
11. Filipovic J, Hozzová J, Nezarat A, Olha J, Petrovič F (2021) Using hardware performance counters to speed up autotuning convergence on gpus, *ArXiv*
12. de Oliveira Castro P, Kashnikov Y, Akel C, Popov M, Jalby W (2014) Fine-grained benchmark subsetting for system selection, In: Proceedings of Annual IEEE/ACM International Symposium on CGO, ser. CGO '14, NY, USA, p 132–142
13. Balaprakash P, Gramacy R, Wild S (2013) Active-learning-based surrogate models for empirical performance tuning, Proceedings - ICCC, pp 1–8, 09
14. Wang Z, O'Boyle MF (2009) Mapping parallelism to multi-cores: a machine learning based approach, In: PPOPP Proceedings of the 14th ACM SIGPLAN, pp 75–84
15. Qawasmeh A, Malik AM, Chapman BM (2015) Adaptive openmp task scheduling using runtime apis and machine learning, In: 2015 IEEE 14th international conference on machine learning and applications (ICMLA). IEEE, pp. 889–895
16. Jäntschi L, Bolboaca S-D (2005) Pearson versus spearman, kendall's tau correlation analysis on structure-activity relationships of biologic active compounds. Leonardo Electron J Practices Technol 6:76–98
17. McCalpin JD (1995) Stream: Sustainable memory bandwidth in high performance computers, Link: www.cs.virginia.edu/stream/
18. Yuki T (2014) Understanding polybench/c 3.2 kernels, In: International workshop on Polyhedral Compilation Techniques (IMPACT), pp. 1–5
19. Yuki T, Pouchet L-N (2015) Polybench 4.0, accessed: April 21 2020. [Online]. Available: https://www.cs.colostate.edu/AlphaZsvn/Development/trunk/mde/edu.csu.melange.alphaz.polybench/polybench-alpha-4.0/polybench.pdf

20. Klambauer G, Unterthiner T, Mayr A, Hochreiter S (2017) Self-normalizing neural networks, In: Advances in neural information processing systems, pp 971–980
21. Martínez A, Sikora A, César E, Sorribes J (1970) Elastic: A large scale dynamic tuning environment, Scientific Programming, vol. 22,
22. Sikora Morajko A, Caymes-Scutari P, Margalef T, Luque E (2007) Mate: Monitoring, analysis and tuning environment for parallel/distributed applications. Concurrency Comput: Pract Exp 19:1517–1531
23. Bailey DH, Barszcz E, Barton JT, Browning DS, Carter RL, Dagum L, Fatoohi RA, Frederickson PO et al (1991) The nas parallel benchmarks. Int J Supercomput Appl 5(3):63–73
24. César E, Moreno A, Sorribes J, Luque E (2006) Modeling master/worker applications for automatic performance tuning. Parallel Comput 32:568–589
25. Kotsiantis S, Kanellopoulos D, Pintelas P (2005) Handling imbalanced datasets: A review. GESTS ICSSE 30:25–36
26. Batista GEAPA, Prati RC, Monard MC (2004) A study of the behavior of several methods for balancing machine learning training data. SIGKDD Explor Newsl 6(1):20–29
27. Kovács G (2019) Smote-variants: A python implementation of 85 minority oversampling techniques. Neurocomputing 366:352–354
28. Elkan C (05 2001) The foundations of cost-sensitive learning, In: Proceedings of the 17th international conference on artificial intelligence, vol. 1,
29. Biau G (2010) Analysis of a random forests model. JMLR 13:05
30. Lorena A, Carvalho A, Gama J (2008) A review on the combination of binary classifiers in multiclass problems. Artif Intell Rev 30(1–4):19–37
31. Gangwar AK, Ravi V (2019) Wip: Generative adversarial network for oversampling data in credit card fraud detection, in Information Systems Security

Authors and Affiliations

Jordi Alcaraz[1] [ID] · **Ali TehraniJamsaz[2]** · **Akash Dutta[2]** · **Anna Sikora[1]** [ID] · **Ali Jannesari[2]** [ID] · **Joan Sorribes[1]** [ID] · **Eduardo Cesar[1]** [ID]

Ali TehraniJamsaz
tehrani@iastate.edu

Akash Dutta
adutta@iastate.edu

Anna Sikora
anna.sikora@uab.cat

Ali Jannesari
jannesari@iastate.edu

Joan Sorribes
joan.sorribes@uab.cat

Eduardo Cesar
eduardo.cesar@uab.cat

[1] Universitat Autònoma de Barcelona, Bellaterra, Spain

[2] Iowa State University, Ames, USA