Towards Automatic Generation of Performance Models for Dynamic Tuning using Machine Learning

Jordi Alcaraz Rodriguez

Universitat Autònoma de Barcelona

Escola d'Enginyeria

Departament d'Arquitectura de Computadors i Sistemes Operatius

Towards Automatic Generation of Performance Models for Dynamic Tuning using Machine Learning

Thesis submitted by **Jordi Alcaraz Rodriguez**, belonging to the research line of High Performance Computing for the degree of Philosophiae Doctor in Computer Science, by the Universitat Autònoma de Barcelona, under the supervision of Dr. Anna Sikora and Dr. Eduardo César.

Barcelona, November, 2021

Towards Automatic Generation of Performance Models for Dynamic Tuning using Machine Learning

Thesis submitted by **Jordi Alcaraz Rodriguez** for the Doctor of Philosophy degree in Computer Science. This doctoral thesis belongs to the High Performance Computing research line, performed in the Computer Architecture and Operative Systems department in Universitat Autònoma de Barcelona, under the supervision of Dr. Anna Sikora and Dr. Eduardo César.

Date: 21st of November, 2021

PhD Student: Firmado por ALCARAZ RODRIGUEZ, JORDI (FIRMA) el día 22/11/2021 con un

Co-Directors: ANNA BARBARA SIKORA   CESAR GALOBARDES, EDUARDO

This work has the support of the Ministerio de Economía y Transformación Digital MINECO-SPAIN under contracts TIN2014-53234-C2-1-R and TIN2017-84553-C2-1-R, MICIN-Spain under contract PID2020-113614RB-C21 and by the Generalitat de Catalunya GenCat-DIUiE (GRR) project 2017-SGR-313.

Resumen

En los sistemas actuales son necesarios nuevos enfoques para generar modelos de rendimiento debido a la heterogeneidad. Una alternativa a los modelos analíticos tradicionales podría ser el uso de algoritmos de aprendizaje automático, ayudando a la creación automática de modelos de rendimiento para predecir la configuración correcta para diferentes parámetros de la aplicación.

Para poder crear modelos de rendimiento, las métricas se utilizan como entradas para calcular o seleccionar los valores adecuados para uno o varios parámetros que pueden afectar al rendimiento. La correcta selección de las métricas es importante debido a que la información puede ser redundante o insuficiente. Además, se deben tener en cuenta múltiples escenarios a la hora de generar modelos, como diferentes tamaños de problema, para obtener el comportamiento en diferentes condiciones, lo que permite generalizar las relaciones entre métricas y evitar relaciones adaptadas a un solo escenario.

En esta tesis abordamos los dos problemas previamente explicados para aplicaciones multihilo utilizando OpenMP con el desarrollo de dos metodologías.

En primer lugar, se desarrolla una metodología para encontrar el conjunto adecuado de métricas con el fin de caracterizar el comportamiento de una región paralela. Mediante el uso de esta metodología se reduce el número de métricas necesarias para caracterizar correctamente una aplicación o una región paralela, disminuyendo la sobrecarga al medir todas las métricas necesarias. Hemos decidido utilizar contadores hardware de rendimiento como métricas para caracterizar la ejecución de regiones paralelas OpenMP. Utilizando esta metodología, el número necesario de contadores hardware se redujo a menos de la mitad de la lista de uso general de contadores disponibles, evitando al mismo tiempo la pérdida de información.

La segunda metodología se desarrolla para construir un conjunto representativo y equilibrado de patrones disponibles en aplicaciones paralelas. Dado un conjunto de regiones paralelas candidatas que se incluirán en un conjunto para el ajuste del rendimiento, cada candidato se compara con los patrones ya incluidos en el conjunto para determinar si cubren, o no, una región diferente del espacio de búsqueda. Esta comparación se basa en el análisis de correlación de las métricas medidas para el candidato. Por ejemplo, en uno de los sistemas probados, se generó un conjunto con solo 8 patrones de 33 candidatos extraídos de los benchmarks STREAM y PolyBench.

El conjunto generado se desequilibra cuando se utiliza para ajustar el rendimiento porque en un sistema los valores de algunos parámetros generalmente proporcionan un mejor rendimiento que otros valores. En consecuencia, los algoritmos de aprendizaje automático pueden tener un rendimiento inferior debido a casos subrepresentados. Por lo tanto, técnicas para contrarrestar el desequilibrio natural son necesarias.

Se proporciona un estudio inicial para encontrar los algoritmos de aprendizaje automático con una mejor precisión para ajustar el número de subprocesos. El estudio incluye: a) métodos de datos para equilibrar el conjunto para el parámetro objetivo; b) métodos algorítmicos para modificar la forma en que se calcula el error; y c) métodos ensemble, combinación de múltiples modelos en uno único, proporcionando una hipótesis general de cada modelo individual.

Summary

New approaches are necessary to generate performance models in current systems due the heterogeneity found in new systems. An alternative to traditional analytical models could be the use of machine learning algorithms, which may help to automatically create performance models to predict the correct configuration for one or multiple application's parameters.

To be able to build performance models, metrics are used as inputs to calculate or select the proper values for one or multiple parameters which can impact performance. The selection of the correct metrics is important as information can be redundant or insufficient. In addition, multiple scenarios should be taken into consideration when generating models, such as different problem sizes, to obtain the behaviour under different conditions, which allows to generalize the relationships between metrics and avoid relationships tailored to only one scenario.

In this thesis we tackle the two previously explained problems for multi-thread applications using OpenMP with the development of two methodologies.

First, a methodology to find the proper set of metrics for characterizing the behaviour of a parallel code region is developed. Through the use of this methodology the number of metrics necessary to correctly characterize an application or a code region is reduced, decreasing the overhead when measuring all the necessary metrics. We have decided to use hardware performance counters as metrics to characterize the execution of OpenMP parallel regions. Using this methodology the number of hardware performance counters was reduced to less than half the available general purpose list of available counters while avoiding loss of information.

The second methodology is developed to build a representative and balanced dataset of patterns found in parallel applications. Given a set of candidate parallel regions to be included in a dataset for performance tuning, each candidate is compared against the patterns already included in the dataset to find whether they cover, or not, a different region of the search space. This comparison is based in the correlation analysis of the metrics measured for the candidate. For example, in one of the tested systems, a dataset was generated with only 8 patterns from 33 parallel kernels extracted from STREAM and PolyBench benchmarks.

The previously generated dataset becomes imbalanced when used for performance tuning because in a system some parameters' values generally provide better performance than other values. Consequently, machine learning algorithms may under-perform due to underrepresented cases and techniques to counter the natural imbalance are necessary.

An initial study is provided to find which machine learning algorithms provide better accuracy for tuning the number of threads. This study includes: a) data methods to balance the dataset for the target parameter; b) algorithmic methods to modify how the error is calculated; and c) ensemble methods, the combination of multiple models into a bigger one, providing a general hypothesis from each individual model.

Acknowledgements

I would like to thank the following people who have helped me to complete my research.

First, I would like to thank Ania and Eduardo, who have been the advisors of my research. I thank them for their help and guidance all these years, which have allowed me to discover and tackle some issues which were difficult to find out. Furthermore, they have helped in my research with proposals and providing interesting articles with interesting alternatives from similar works or works in other areas that could be inspiring for the work performed in the thesis, allowing my research to advance more smoothly and be more fruitful.

In the same way, Joan, Josep and Andreu who have been in our, more or less, weekly meetings explaining the development in their research, and also providing suggestions and pointing to possible problems they have encountered previously which may also appear in my research.

I also want to give my thanks to all the people belonging to our department (CAOS), be it teachers who have helped and given support in different ways, or support members like Dani and Gemma which allow everything to work as smoothly as the university allows.

Moreover, I would like to thank both the former PhD students which are now doctors and the ones currently doing their research, as we shared our workplace, our problems and helped each other. I also want to wish good luck to the current students with their research so they become doctors soon.

Additionally, we have collaborated with Iowa University, so I thank Ali Jannesari for his collaboration with us where a lot of ideas came out. Also, thanks his PhD students, which helped to correctly write my last publications, making them more interesting and easier to be accepted by reviewers.

Finally, I wish to thank Sameer Shende for allowing me to collaborate at the Oregon University during summer. It was an interesting experience where I worked with TAU's research group and I met Wyatt, Kevin and Chad, who helped and because each research group has different points of view, it allowed me to learn different approaches to find and solve problems. Thank to Vickie for helping resolve problems related to temporarily living in Eugene.

Contents

List of Figures

List of Tables

Abstract

Nowadays, due to the increase of demand in computation power and the limitations in energy consumption, systems used in high performance computing (HPC) are becoming more and more complex. Newer computation nodes need more computational power, which is solved with the use of additional resources, such as increasing the number of cores in a processor. Additionally, accelerators such as GPUs are becoming popular, but their architecture is different from traditional processors, increasing the system's heterogeneity.

The efficient use of the available resources in a system is necessary to improve applications' performance, making performance analysis and tuning an important topic in HPC. Due to the increase in resources, and also heterogeneity found in systems, adequate analytical models for performance improvement can be very difficult to generate. In addition, the difficulty is further increased as multiple accelerators with different characteristics can be found in newer systems.

New approaches are necessary to generate performance models in current systems. An alternative to traditional analytical models could be the use of machine learning algorithms, which may help to automatically create performance models to predict the correct configuration for one or multiple application's parameters.

To be able to build performance models, metrics are used as inputs to calculate or select the proper values for one or multiple parameters which can impact performance. The selection of the correct metrics is important as information can be redundant or, in the worst case, insufficient to correctly describe the relationship between metrics and performance parameters.

In addition, multiple scenarios should be taken into consideration when generating models, such as different problem sizes, in order to obtain the behaviour of a system under different conditions. With the use of multiple scenarios, the model should be able to generalize relationships between metrics and avoid relationships tailored to only one scenario.

In this thesis we tackle the two previously explained problems for multi-thread applications using OpenMP (de facto standard for multi-threaded applications), with the development of two methodologies.

First, a methodology to find the proper set of metrics for characterizing the behaviour of a parallel code region is developed. Through the use of this methodology the number of metrics necessary to correctly characterize an application or a code region is reduced, decreasing the overhead when measuring all the necessary metrics. We have decided to use hardware performance counters as metrics to characterize the execution of OpenMP parallel regions. Using this methodology, the number of hardware performance counters was reduced to less than half the available general purpose list of available counters while avoiding loss of information.

The second methodology is developed to build a representative and balanced dataset of patterns found in parallel applications. Given a set of candidate parallel regions to be included in a dataset for performance tuning, each candidate is compared against the patterns already included in the dataset to find whether they cover, or not, a different region of the search space. This comparison is based in the correlation analysis of the metrics measured for the candidate. For example, in one of the tested systems, a dataset was generated with only 8 patterns from 33 parallel kernels extracted from STREAM and PolyBench benchmarks.

When the previously generated dataset is used for performance tuning, an imbalance problem appears as the targets are now performance parameters instead of representative code regions. This imbalance appears because in a system some parameters' values generally provide better performance than other values for most cases. Consequently, machine learning algorithms may under-perform due to underrepresented cases, making the use of techniques to counter the natural imbalance necessary.

An initial study is provided to find which machine learning algorithms provide better accuracy for tuning the number of threads. This study includes: a) data methods to balance the dataset for the target parameter; b) algorithmic methods to modify how the error is calculated when a prediction is incorrect, such as applying errors inversely proportional to the frequency of each value of the tuning parameter; and c) ensemble methods, which are the combination of multiple models into a bigger one, providing a general hypothesis from each individual model.

Summarizing, we have developed a method for deciding the minimum amount of hardware performance counters necessary to characterize the execution of an OpenMP code region on a given architecture, minimizing in this way the overhead for obtaining those metrics. Then, we have also developed a methodology for creating balanced and representative dataset of code regions executions, which is one of the basic requirements for successfully applying ML methods. Finally, we have explored most of the available methods for automatically generating performance models that are able to manage the natural imbalance appearing in performance tuning.

Keywords: performance analysis; performance analysis tool; performance tuning; performance models; artificial intelligence; machine learning; performance dataset generation; generative adversarial networks; imbalanced dataset; random forest;

Chapter 1

Introduction

This chapter introduces the problems we have tackled in this thesis focused on the field of performance tuning of parallel applications. First, the motivation behind this work is discussed, followed by the objectives we plan to accomplish in this thesis. Afterwards, the main contributions brought about with the development of this thesis are explained and, finally, the organization of this work is presented.

1.1 Motivation

Nowadays, due to the increase of demand in computational power because of the development of more complex simulations or new problems to solve in different fields, more advanced systems are developed in high performance computing (HPC) with more computational power to fill the demand. Consequently, an endless cycle seems to be generated between the generation of more complex applications and increases in computational power.

This increase in demand is satisfied with an increment on the resources in systems, such as increasing the number of cores in a processor and the use of systems with multiple processors. In this way, in 2005 Intel Xeon processors had 2 cores and 4 threads, and now, Intel Xeon W-33754 has 38 cores with 76 threads. In the case of AMD's EPYC 3, it offers some processors with 64 cores and 128 threads. Additionally, due to energy consumption limitations, systems are becoming heterogeneous with the use of technologies such as dynamic clock speed for each core in a processor. Moreover, there are processors with hybrid architectures, as can be seen in ARM processors, where there are cores for complex workloads with high clock speed and cores for simpler tasks with lower clock speeds. Furthermore, there are new resources which can be used alongside processors, they are known as accelerators, and have a different structure than traditional processors, which further increases systems' heterogeneity.

In order to help determine the sources of performance problems and their possible solutions, modeling performance is important to find an abstract representation of the system [1]. This abstract representation takes different metrics available in the system as inputs, obtaining parallel code regions' representations (signatures) characterizing the behaviour in different parts constituting the system. Signatures are constituted by values, such as memory accesses at each memory level, execution time, number of executed instructions, among others in a wide range of possible events to measure.

Because of the additional hardware in new systems, there is also an increase in the metrics available and the appropriate metrics should be selected. In modern systems, there is a high number of metrics available, for example, with the use of PAPI [2], AMD's EPYC 7551 reports 136 available metrics and multiple metrics mention more than 15 possible flags. This high number of metrics generates a huge amount of relationships between the different metrics and their potential impact in performance.

Additionally, multiple parallel programming paradigms are used in one application with the objective of sharing work between multiple systems (multi-threading with memory sharing inside the system and the use of messages to communicate between different nodes), generating new performance problems to tackle. Consequently, metrics for each paradigm must be measured, increasing the number of necessary metrics to accurately describe the behaviour of an application. In the case of sharing work between multiple systems, the same metric may be measured in each individual system and additional metrics which characterize communication and issues tied to communication between all nodes must be collected. Therefore, finding the problems' sources is more difficult because there are now more elements which have to cooperate efficiently and avoid wasting resources. Besides, there are more parameters which can be modified to improve performance.

As a result of both the increase in resources and the use of multiple parallel programming paradigms, performance tuning demands more complex models when analyzing performance, which makes the creation of models by experts a very difficult task to accomplish. Therefore, new approaches are necessary in order to facilitate the creation of performance models or for their automatic generation [3][4][5][6].

Accordingly, an idea is to consider how currently the use of machine learning is growing. Machine learning can be considered the state of the art for many problem where huge amounts of data need to be analyzed. Therefore, the challenge of generating performance models may be solved applying machine learning techniques.

The behaviour of a system could be modeled with machine learning algorithms. Then, the generated model could be used to further help experts in the creation of future performance models. Furthermore, the ML model could also automatically generate a performance model which is able to provide predictions of the proper value, or an approximation, for tunable parameters to improve performance in parallel applications.

However, there is a fundamental principle known as *"garbage in, garbage out"* [7][8][9] when using datasets to build models with machine learning. This principle states that the generated model can only be as good as the data used in its training. Therefore, the right data (a valid dataset with the proper inputs) from which proper relations can be extracted and samples from multiple scenarios must be correctly represented when building models with ML. As an example, if a user wants to build a model to detect if there is a bottleneck in L3 cache: a) metrics related to L3 should be included in the model; b) scenarios where bottleneck in L3 cache appear must be included in conjunction to scenarios where they do not appear.

Therefore, in order to apply machine learning, first we need to define the inputs of the model and secondly an adequate dataset is necessary when the model is trained. Consequently, there are two main problems to address: 1) a big number of metrics can be measured in parallel applications, **how can the execution of an application be properly represented with a limited number of metrics?** 2) representative parallel regions (parallel patterns) executed in different scenarios should be used to generate a dataset, **how can a representative and balanced dataset be built?**

To simplify the problem, we have decided to limit the scope of our work to OpenMP multi-threaded applications. For these applications, hardware performance counters can be used to generate signatures and obtain a representation of the behaviour of parallel code regions, but due to the high number of hardware performance counters available in a processor, a way to either discriminate or simplify the number of metrics is needed to reduce the number of inputs a model needs to analyze performance. Furthermore, when analyzing an application and applying dynamic tuning, which makes changes in parameters at execution time, the list of metrics should be as small as possible to reduce overhead but, at the same time, without losing important information about the behaviour of the application, as both overhead or loss of information could aggravate the difficulty of analyzing performance.

Additionally, multiple parallel regions should be considered in order to conveniently cover the search space, but, at the same time, avoiding the over/under-representation of any region. These regions are considered representative patterns. Furthermore, multiple scenarios should be executed for the correct generation of a representative dataset with the selected inputs. Otherwise, the model could infer relationships which may only appear under certain conditions. In the case of OpenMP applications, the multiple scenarios could be constituted using multiple combinations of parameters such as problem sizes, number of threads or thread affinities.

Finally, it can happen that even though the dataset is built correctly, it can become imbalanced. In performance tuning, certain parameter values are predominant in a majority of scenarios due to the system's characteristic. As a consequence, the dataset can be naturally imbalanced, so balancing techniques are necessary to generate appropriate models. Balancing techniques can be applied directly into the dataset with either the removal or replication of samples, or with techniques that generate synthetic samples. Some balancing techniques which do not modify the dataset are algorithmic modifications in some machine learning algorithms and ensemble models.

Summarizing, machine learning seems to be a promising approach to automatically generate performance models for complex systems without the intervention of performance experts. However, in order to generate accurate performance models with machine learning, it is necessary to: 1) select the adequate set of features describing performance; 2) create representative and balanced datasets; and 3) analyze which is the machine learning techniques that best adapt to the problem characteristics.

1.2 Objectives

The main objective of this thesis is to research whether machine learning techniques can be used to automatically generate performance models. The model generated without the intervention of performance experts would be used for performance tuning of an application or code regions inside an application.

First, we need to use metrics to characterize the behaviour of parallel code regions [10]. This characterization could be based on the signature of a parallel region, composed of different hardware performance counters which describe the operations performed in a system. However, there is the previously described *"garbage in, garbage out"* problem; consequently, proper approaches to select input metrics and also for the selection of the different scenarios needed to build datasets are necessary, so that relations between the inputs and the objective of the model can be correctly identified. Then, a methodology to discriminate which signatures are necessary to build a balanced and representative dataset of parallel regions is necessary. At the end, a model with machine learning techniques such as artificial neural networks or decision trees, could be built with the objective of either describing the behaviour of the code region's signature or to directly provide the parameters' values which need modifications in an application to improve performance.

To generate performance models with machine learning, there are some objectives which need to be fulfilled.

First, it is necessary to select the metrics composing the signatures characterizing the behaviour of a code region. There is a wide range of available metrics in the system, each describing the behaviour of a certain part of the system, but measuring all metrics generates overhead both in time and memory which could be reduced if less metrics are accessed. **The first objective is to provide a methodology to reduce the number of metrics available in the system while minimizing redundancy in the information provided by the metrics without the loss of any useful information.**

Secondly, machine learning methods need a proper set of data to be trained, as the generation of a valid model is only possible if the data used for training is able to provide the necessary information. **Thus, the second objective is to create a methodology to generate valid datasets where code regions found in parallel applications are represented.** Each code region should cover a unique portion of the N-dimensional space represented by the measured hardware performance counters. Moreover, the dataset should be balanced to avoid problems in training due to over-represented classes, in this case, parallel code regions that are representative of the behaviour of different code regions appearing in parallel applications.

Finally, the generated dataset should be used to train machine learning models which provides a configuration of tunable parameters, allowing to improve performance of parallel applications. Although the dataset has been generated in a balanced way, the dataset can appear imbalanced when it is used for performance tuning. This problem appears because the best value for a parameter that impacts performance cannot be known a priori, otherwise there wouldn't be the need to create performance models. In addition, a limited number of values for a parameter are generally the best for most cases. **Consequently, the third objective is the creation of models with machine learning techniques for performance analysis and how to deal with the natural imbalance which can appear in datasets for performance tuning.**

A general overview of the objectives of this thesis to automatically generate performance models can be seen in Figure 1.1:

- Design a methodology to generate signatures of parallel code regions which is able to detect and discard metrics with redundant information.
- Generate a balanced and representative dataset of parallel code regions using signatures generated by the previous methodology.
- Develop models with machine learning for performance analysis and tuning with a balanced and representative dataset.

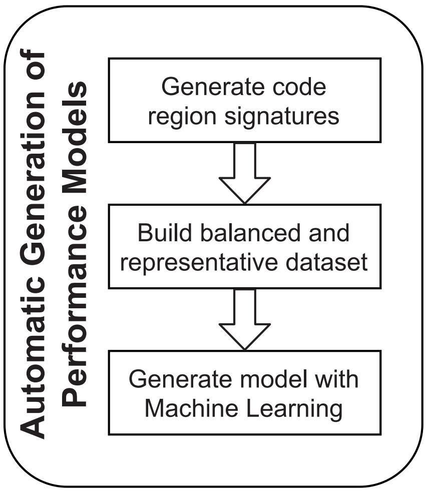

Automatic Generation of Performance Models
Generate code region signatures
Build balanced and representative dataset
Generate model with Machine Learning

Figure 1.1: General overview.

1.3 Main contributions

The main contribution of this thesis is to achieve the objective presented in the previous section: **show that machine learning models can be used to automatically generate valid performance models for performance tuning.** Considering that the scope has been limited to parallel applications using OpenMP to be able to advance our research in a reasonable time, this thesis presents the following three main contributions:

- **Signature reduction.** The first contribution of this thesis is the creation of a methodology to find the proper set of metrics in a system. Redundant metrics are found using a correlation analysis between pairs of metrics. High correlation values indicate the possibility of two metrics being redundant, suggesting a candidate to be discarded. This methodology should be used with care as high correlation between two metrics does not mean causation between them. Therefore, the relationship between both metrics should be checked logically to verify whether the metrics are indeed redundant or not. For OpenMP applications, hardware performance counters are used as metrics.
- **How to build datasets for performance tuning.** The second contribution is a methodology to generate balanced and representative datasets for performance tuning. This methodology generates a dataset, called pattern collection, where the signatures of parallel code regions for multiple problem sizes are gathered to represent the behaviour of a pattern in multiple memory levels of the hierarchy. Each pattern included in the dataset should cover a unique portion of the space represented by the hardware performance counters composing the signatures. A kernel is considered a pattern if its behaviour is not found to be similar to any already included pattern in the collection. Similarity is analyzed using correlation analysis with both Spearman and Kendall's Tau.
- **Dealing with naturally imbalanced datasets.** After the selection of metrics and the generation of the dataset, machine learning techniques must be applied to automatically generate performance models. However, due to the natural distribution in the values of performance parameters, the dataset may become imbalanced. Therefore, the third contribution is to study which machine learning algorithms can generate adequate performance models and which techniques can be applied to counter the imbalance found in the dataset.

1.4 Thesis Organization

This thesis is organized in the chapters described bellow.

Chapter 2: Related Work explains related works in the field of performance tuning and additional approaches where machine learning is used to help in performance analysis and/or tuning.

Afterwards, **Chapter 3: Background** presents methods and techniques used in this thesis to accomplish the objectives introduced previously.

Chapter 4: Signature Reduction deals with the problem of selecting the appropriate metrics to obtain a representation of the behaviour of a parallel application or a parallel code region. Because we limited the work to OpenMP, the characterization of the behaviour is done with hardware performance counters. The developed methodology deals with the detection of redundant metrics. Consequently, reducing the number of metrics needed when building the signature of a parallel code region without information loss.

Next, **Chapter 5: Building Datasets for Performance Tuning** explains how to build a balanced and representative dataset. This dataset is called pattern collection and is later used to train machine learning models. The pattern collection contains the characterization of different patterns executed under different conditions, such as number of threads and their affinity, and problem size. The characterization is the signature of an execution built with the hardware performance counters obtained after applying the methodology previously presented in Chapter 4.

Then, **Chapter 6: Dealing with Naturally Imbalanced Datasets** explains how to use the pattern collection, which was created in Chapter 5, to generate performance models using machine learning. In this chapter, the previously generated dataset became imbalanced due to the nature of tuning parameters, which show a predilection towards some configuration values. As a consequence, the dataset becomes imbalanced when it is used to find an ideal, or a possible optimum, for the configuration of a parameter which needs to be modified in an application to improve performance or to solve performance problems.

Finally, **Chapter 7: Conclusions and Future Work** summarizes the work done in this thesis and the results obtained after the presented methodologies are applied. Moreover, some open lines are explained with the objective of, in the future, improving both the methodologies and performance models generated using the techniques explained in this thesis.

Chapter 2

Related Work

This chapter explains the state of the art of performance modeling for parallel applications, which we divided into three parts: analytical performance modeling, search based auto-tuning and machine learning based approaches.

First, some approaches for analytical performance modeling are presented. Analytical models are built with the help of experts to infer relationships between metrics and performance.

Then, the search based auto-tuning approach used in some frameworks is explained. Given an application and a list of possible values for different parameters, modifications in the application are performed applying a search algorithm to the n-dimensional space generated by combining all the possible parameter values. The objective is to search an optimal configuration of parameters which improve performance without exploring all the possible configurations.

Finally, some machine learning based approaches are presented. These approaches replace the need of experts because the inference of relationships in heterogeneous systems is a very complex task, generating the need of alternative methods to find performance issues. Machine learning is used to aid in performance tuning or to build models which predict optimal parameters' configurations.

Our work is similar to the machine learning based approaches. We want to use machine learning to generate performance models to find ideal configurations for tunable parameters in parallel code regions. However, the presented approaches skip important parts of the process, such as the selection of metrics and dataset generation, which are critical when using machine learning.

2.1 Analytical performance modeling

In this section different analytical performance models and analytical methodologies for performance tuning are explained. Some of these models may be outdated as they do not take into account paradigms found in new systems which may generate heterogeneity.

2.1.1 Performance model for OpenMP memory bound multisocket

In [11] a performance model is presented to predict execution time and L3 cache misses. The model is used for OpenMP memory bound applications in multisockets systems.

This model takes as metrics the total number of last level cache misses and execution time in one processor. These metrics are obtained from 1 core to the maximum number of cores in a single processor and are used to characterize memory concurrency.

Then, two factors describing concurrency as the number of cores in use increases are calculated, one factor representing overheads in cache misses and another representing latency overheads due to cache misses. These two factors are used to estimate L3 cache misses and execution time for different thread affinities when more than one processor is used.

This model describes two cases depending on accesses to memory: a) an ideal case where memory accesses are parallel; and b) a case where accesses are serialized.

In [12] the same authors presented a performance model for OpenMP memory bound applications based on memory footprint. The objective of this model is to estimate execution time and performance degradation due to memory contention. This model has the assumption that memory contention in the last cache level is the main cause of performance degradation and that the system is homogeneous.

Trace generation is used with small workloads to extract the memory footprint for parallel code regions. For each memory access the following metrics are extracted: the access type, the virtual memory address, data size in bytes and number of non-memory instructions to the previous memory access. Afterwards, the memory footprint for unknown workloads is estimated using execution information such as hardware performance counters.

2.1.2 Dynamic workload balancing

In [13] a methodology for dynamic workload balancing of data-intensive applications for homogeneous clusters is presented. In some applications which explore the same data multiple times, data can be partitioned so it can be accessed in parallel to increase performance.

The objective of this methodology is to determine a partition factor and dynamically establish the order in which chunks of data are scheduled. Additionally, the number of processing nodes can be dynamically adapted. This work uses efficiency, computational and communication time, and memory usage as performance metrics.

The methodology is divided in three main steps:

- Data set partitioning. The characteristics of the nodes (such as RAM memory and cache sizes) are used to establish different partitioning factors the data is divided with.

- Load balancing scheduling policy. First, the application is executed with a high number of workers and a big data set partition factor. Monitoring is used to find the execution time associated to each data chunk and stored to have historical execution data. Then, when historical data is available, data chunks are executed in descending order, from the one with highest execution time to the one with the lowest.
- Adapting number of resources used. The efficient number of workers is calculated with an index relating the estimated execution time and the efficient use of resources. Efficiency is calculated as a relation between the mean computation time of each data chunk and the availability time of a node.

2.1.3 Dynamic Pipeline Mapping

In [14], a methodology to dynamically improve performance in pipeline applications, called Dynamic Pipeline Mapping (DPM), is presented. The objective of this method is to free computational resources assigning consecutive fast stages to the same processor. Then, if possible, slow stages are replicated to the freed resources with the objective of increasing throughput.

An algorithm is used to balance the load in pipeline applications and obtain a solution as closest as possible to the optimal.

Additionally, two performance models are generated: a) a performance model to get an approximated computational time for stages with high computational requirements, which, using a Master/Worker scheme, are replicated into the freed processors to process data in parallel; b) a performance model to measure execution time for stages with low computation requirements, which are grouped and executed consecutively in the same processor.

In [15], DMP is extended to work in heterogeneous systems because the performance model generated by DPM does not work well in heterogeneous systems. This new approach, called Heterogeneous Dynamic Pipeline Mapping (HeDPM), includes the computational capacities in different systems and communication costs.

HeDPM sorts pipeline stages by their computational load and communication requirements. Then stages are matched to resources based on their capabilities.

2.1.4 Dynamic Tuning for the number of workers in Master/Worker applications

A methodology to tune the number of workers in Master/Worker applications is presented in [16]. The objective of the model is finding an ideal number of threads to partition the workload efficiently to avoid idle nodes or a saturation due to excessive communications. The methodology only takes into consideration homogeneous Master/Worker applications, such applications are composed of tasks which are similar in size and computational requirements.

The performance model to calculate the number of workers uses MATE [17] to monitor the application and apply the changes dynamically. A parameter defining the number of workers is modified through MATE to tune the application. The metrics to monitor are the overhead to send a message, communication time required per byte, task sizes, result sizes and execution time per task. These metrics are then used to build different performance functions to find the ideal number of workers.

2.1.5 Performance model for GPU architectures

An analytical model for GPU architectures is presented in [18] with the objective of predicting execution time. The authors of this work consider that execution time in GPUs is dominated by the cost of memory operations.

Execution time is calculated using two metrics defined by the authors: a) Memory Warp Parallelism, representing an estimation of the maximum number of memory requests which can be performed in parallel; and b) Computation Warp parallelism, representing the computation which can be performed by warps while another is waiting to access memory. Then, both metrics jointly with waiting cycles, computation cycles per warp, number of memory instructions, and active number of SMs are used to model execution time.

In [19] a power and performance prediction model is built to select the number of streaming multiprocessors to use in a GPU for both power and temperature reduction. The previous model is used to predict execution times.

In this case the performance per watt is used to select the optimal number of streaming multiprocessors. The ideal number maximizes the following ratio: work divided by execution time for N multiprocessors, then divided by the power when using N multiprocessors.

2.1.6 Diogenes and Feed Forward Performance Model

Diogenes is a tool to identify synchronizations and memory transfers that are problematic [20], for example unnecessary synchronizations or duplicated memory transfers.

Diogenes uses a new monitoring approach called Feed Forward Performance Model (FFM). FFM allows multi-stage/multi-run instrumentation, adjusting instrumentation depending on the behaviour of an application.

There are five stages in the FFM model:

- Stage 1 - Baseline Measurement. Execution time is measured for analysis in future stages. Additionally, stack traces are recorded to identify synchronizations with the GPU.
- Stage 2 - Detailed Tracing. Stacktrace and times (synchronization and/or total) are collected for all synchronizations and memory transfer operations.

- Stage 3 - Memory Tracing and Data hashing. In this stage problematic operations are detected, problematic operations are defined as those that can be removed or moved to another location, obtaining an improvement in performance, while avoiding problems in the application's correctness. FFM determined whether synchronizations are really necessary checking if the protected data is accessed afterwards or not, in the case the protection is not necessary, the synchronization may be removed to reduce execution time and improving performance. Problematic memory transfers (redundant memory copies) are detected with the use of hashed, in the event a call to copy data between CPU and GPU is detected, a hash is generated for the data to copy compared to previously copied data's hash, if there are no changes in the data, the copy may be redundant.

- Stage 4 - Sync-Use Analysis. Timing information between synchronization and data access is collected to determine whether synchronizations are misplaced or not.

- Stage 5 - Analysis Stage. The execution of an application is modeled with the use of a graph, which could be considered as two Gantt Charts joined by edges. In this graph nodes are events, edges in the same processor are times and edges between CPU and GPU are communications. Each node has four attributes, which identify their type, start time, if a problem was detected in stages 3 and 4, and time between synchronization and data usage. Edges are labeled with the duration of each event. Three types of problems are modeled, which are unnecessary synchronization, misplaced synchronization and unnecessary memory transfer.

In [21] an extension was performed to the Feed Forward Performance model, the main changes are applied in stages 2 and 5.

In stage 2, memory allocations and free operations are included in the tracing to enable the construction of dataflow graphs. In the case of stage 5, a list is generated with solutions to problems discovered by the model. The memory graph generated in stage 2 and the list of problematic operations help determine the possible solutions.

2.2 Auto-tuning with search in the tunable parameters' space

There are some tools which instead of working with performance models, use a search algorithm in the space generated by the possible values of the tunable parameters. These tools perform an iterative search modifying the tunable parameters according to the search algorithm employed. After the changes in parameters are applied, performance is measured. Then, this search is repeated until the best configuration obtained is found, which may be the ideal or a sub-optimal configuration if an exhaustive search is not performed [4].

In this category tools such as Active Harmony [22] which is able to dynamically modify which libraries are in use, modify function calls to apply other algorithms or change an application's parameters to improve performance. Active Harmony initially used a greedy algorithm. Howerver, it is a bad approach to use brute force if there is a large number of possible parameters' combinations, so an alternative search algorithm was used based on the simplex method to find minimums. This search method is described in [23].

Another project in this category is AutoTune [24] which developed the Periscope Tuning Framework (PTF) [25]. PTF is a tool that makes use of plugins to automatically perform a search in multiple parameters to find an optimal configuration. This tool is explained with more detail in the next chapter.

2.3 Performance analysis and/or tuning with machine learning

In this section we explain different projects which use machine learning for analyzing and tuning parallel applications. Machine learning is used as an alternative to analytical models, with the goals of analyzing performance and to improve an application's performance applying modifications to parameters.

2.3.1 READEX

READEX (Runtime Exploitation of Application Dynamism for Energy-efficient eX-ascale computing) [26][27] is an european project that had the objective of developing dynamic auto-tuning tools to improve both performance and energy efficiency in exascale computing for heterogeneous and embedded systems. An additional goal of this project was to apply machine learning to adapt parameters according to the dynamic behaviour in HPC applications.

READEX performs a previous analysis of the application with Periscope Tuning Framework (PTF) and a *representative* data set. This analysis is performed to automatically discover important regions, characterize dynamism and obtain estimates about performance and energy efficiency of the application to be executed later.

To obtain the optimal parameter configurations for a given system, multiple configurations are executed with different search strategies using PTF. After the exploration is performed, the results for various objective metrics are stored in a database. Then, a classifier model is used to classify regions of the application into predefined scenarios. In addition to the classifier, a configuration selector is used to map between scenarios and optimal parameter configurations. A calibration mechanism is used in the case the behaviour of the application does not map to any of the previously known scenarios.

READEX presented a study [28] where neural networks, support vector machines and decision trees are used to predict (for better energy consumption) optimal hardware parameters (number of threads, core and uncore frequency). Moreover, energy consumption is also predicted.

This study was performed for four sparse matrix algorithms. The best generated models were obtained with neural networks and one model was trained for each sparse matrix algorithm. A summary of the results for neural networks, trained with 90% of the data in the training set, are shown in Table 2.1.

Table 2.1: Accuracy in READEX's study

Accuracy(%)	**SpMMadd**	**SpMMmult**	**SpMVmultCSR**	**SpMVmultIJV**
Number of threads	70	70	68	21
Core frequency	96	94	68	97
Uncore Frequency	69	63	59	59
Total energy	89	89	75	91

2.3.2 APARF

APARF (automatic, portable and adaptive runtime feedback-driven framework) is a framework combining low-level tasking runtime APIs, profiling and machine learning to select the optimum task scheduling for an application [29].

Multiple task-based applications from *Barcelona OpenMP Tasks Suite* [30] are used to train the model, as they show different behaviour and different tasks characteristics, considering them a balanced and representative set. Additionally, each application is executed for different problem sizes, thread numbers, compiler flags and task scheduling configurations.

Two aspects are taken into account to manage task scheduling: data locality and load balancing. In the case of load balancing, it is represented by timings in each application thread. On the other hand, data locality is described using hardware performance events related to cache accesses and misses, a ratio between cycles and instructions, and TLB misses.

The machine learning model was built using an Artificial Neural Network to classify the program between three different classes (simple, public and default), each class being tied to a different task pool configuration.

The results of the model are compared against the default task scheduling of different compilers. Only 8 cases from 120 unseen instances were incorrectly classified, where the wrong classifications belong to small problem sizes with inconsistent behaviour. Consequently, the average accuracy achieved is 93%, which is translated into a performance enhancement of 25% compared to the default compiler's configuration.

2.3.3 BLISS

BLISS [4] is a project which employs machine learning in a way similar to auto-tuning with search in the tunable parameters' space, which was explained in Section 2.2.

Bayesian Optimization is used to build models while an application is in execution to avoid offline training. Similar to auto-tuning tools based on parameters' search, it modifies multiple tunable parameters in the application. However, instead of finding a good configuration with a search algorithm, it uses the parameters for each configuration and an output metric (such as execution time) to build a performance model with Bayesian Optimization.

Bayesian Optimization is a method to optimize objective functions (find a minimum). It generates probabilistic models to guide the search of the minimum using input and output values of the function to discover. This method is used when functions are difficult to evaluate for all the possible values [31].

The parameter search in BLISS is guided by the probabilistic model, which is able to suggest application's parameters which help to optimize the objective function, improving the performance of an application.

BLISS' approach is able to find near-optimal parameter configurations, requiring less sampling than approaches based on auto-tuning with search in the tunable parameters' space.

2.3.4 Piecewise Holistic Autotuning with CERE

A project to automatically tune OpenMP applications with a tool named CERE is presented in [32]. CERE decomposes applications into small code regions called codelets which are mapped to OpenMP parallel regions.

In this project codelets are classified using clustering to find codelets with similar performance signatures. CERE combines both static (code characteristics) and dynamic (hardware performance counters and memory bandwidth) metrics to generate performance signatures.

The idea of this project is to apply performance tuning to one codelet in each cluster. Then, the same tuning strategy is applied to codelets belonging to the same cluster. Next, codelets are extracted into different code files and compiled separately with the proper tuning parameters. At the end, a hybrid binary is generated with the application and the separately compiled codelets.

Results for CERE show a reduction of around 6.55$\times$ the time required to evaluate codelets, in comparison to evaluating the whole application, and an accuracy around 93.66% for the NAS benchmark.

2.3.5 Other approaches

Proctor [5] is a semi-supervised framework for anomaly detection in HPC. The framework tries to detect nodes with anomalous behaviour and, in the case an anomaly is detected, it is classified into one of the types known by the model.

Unsupervised learning with autoencoders is used for anomaly detection. A set of metrics are used as the input and the autoencoder generates a reconstruction of the metrics. An anomaly is detected if the error in the reconstruction is higher than a predetermined threshold. Contrarily, supervised approaches are used to classify anomalies by type.

In [33] [34], a framework to profile an application to discover potential parallelism and insert OpenMP annotations in parallelizable loops using machine learning is presented. The framework has two steps:

1. Profiling analysis. The compiler analysis is extended with dynamic information about the application. Additional code is inserted for each variable and memory access with information about memory address and symbol table references to discover data dependencies. Additionally, the control flow of the application is analyzed. This additional information is then used to build a control and data flow graph. This graph is then used to discover potential parallel loops in the application. If a case where possible data dependencies are detected but they do not appear in the profiled execution, the user is informed to select if the parallelization for the loop should be discarded.

2. Annotate OpenMP loops. The generation of parallel code is limited to only OpenMP and *FOR* loops. Read and write operation are analyzed to find data dependencies and decide whether the accessed variables and arrays should be set to private or not. Additionally, necessary synchronizations and reduction operations are recognized. A model is created with machine learning using a multi-class support vector machine to determine if parallelism should be applied to a loop and also its OpenMP scheduling policy. The features used in this model are related to instructions, data accesses, branches and number of iterations in a loop.

The framework achieves on average a 96% of the performance from the original OpenMP versions of the benchmarks extracted from NAS and SPEC.

In [35] an approach to predict the number of threads and the scheduling policy for an application using machine learning is presented. In this work two different models are used, one for each parameter to tune: a scalability model to detect the ideal number of threads built using a feed-forward Artificial Neural Network; and a scheduling policy model generated with a support vector machine.

To predict scalability and scheduling for an unseen program, features about the program are necessary. The selected features are code features (source code features such as operations, control flow and memory accesses) and data features (loop counts and performance counters such as data cache and branch miss rate).

A total of 20 different programs are executed and models are trained using leave-one-out cross-validation. Due to the use of leave-one-out cross-validation 20 different models are trained for each parameter to tune, in each trained model a different program is excluded.

Results show that the models achieve more than 95% of the total performance compared to the ideal configurations in Xeon processors and above 80% for Cell. However, some results are surprising, for example a serial execution using cyclic scheduling is reported to be faster than a serial execution with the default scheduler, with an speed up of around 1.3×.

MaSiF [36], a machine learning approach for auto-tuning of parallel skeletons is presented. This approach uses K nearest neighbors (KNN) and principal component analysis (PCA) to reduce the tuning parameters' search space.

In MaSiF there is a previous phase were the optimum parameters for multiple programs are obtained using a random search in a subset of parameters. The search explores the 10% of the parameters' space according to the experimentation. The measured metrics are also stored and will be used with K nearest neighbors with a value of $K = 3$ when a new program is executed to find to which known programs it is closer to.

Then, the search of the configuration parameters is performed with dimensionality reduction from 5 dimensions (five predefined tuning parameters) to two dimensions using principal component analysis. In this new dimension, the optimal configuration parameters are represented and a transformation is applied to the eigenvectors using the eigenvalues, where authors claim that "we start at the mean and search along 1.5x in the direction of the eigenvectors, which corresponds to covering 99,7% of the variance". This makes no sense as the 1.5 factor appears out of nowhere and also moving the eigenvector provides nothing new as data points may or may not fall in the line formed by the eigenvector. Additionally, no explanation is provided about how to use the principal component analysis with the result from the KNN as in the paper the PCA is a step performed previously.

Discussion of the presented approaches

The previously described analytical performance models are good approaches to solve the problems they tackle under certain conditions. However, with the evolution of systems, which are becoming heterogeneous, their shortcomings are becoming obvious due to their limitations, limitations generally described by their own authors. These models could be adapted for both heterogeneous and dynamical loads, but analytical models require extensive knowledge and time to be generated properly.

In the case of tools, such as auto-tuning with search, there is a lot of potential to solve performance problems. Nonetheless, the overhead introduced into the application to find the proper parameter configuration should be taken into account, as trying different configurations may deteriorate performance and the process can be counterproductive, even if at some point the proper configuration is found.

Therefore, due to the complexity to generate new analytical models such as machine learning, are necessary. Multiple machine learning based approaches for performance tuning have been presented, some of them were developed at the same time as this thesis.

These approaches show that using machine learning is an ongoing and important topic for the automatic generation of performance models without the intervention of performance experts.

In performance modeling the selection of metrics which correctly describe the behaviour of an application is an important issue to take into consideration. In the case of analytical models generated by experts, this selection is limited to the metrics which experts consider the most representative according to their knowledge. This limited selection in analytical models is important in order to reduce the time required when researching relationships between metrics and performance. In machine learning more metrics can be used as the relationships are automatically inferred, allowing the model to infer more knowledge about the behaviour of an application. However, the presented works seem to decide the metrics used for training in the same way as analytical models, using a limited number of metrics the authors consider good for the model to generate, instead of evaluating which of the available metrics should be used.

The use of appropriate sets of parallel regions are necessary to generate a balanced and representative dataset. We were unable to find this important step in any of the approaches which apply auto-tuning with models generated by machine learning. The autotune project with CERE uses clustering to find similar parallel regions in an application. Then, CERE extracts from each cluster a representative member, obtaining for an application a set of balanced and representative parallel regions. The representative parallel regions are used to select tuning strategies to similar code regions.

In our work we want to tackle the problem of how to build balanced and representative datasets for parallel code regions, which is a problem pointed by CERE. Additionally, imbalance due to performance parameters' values and how to mitigate it is important. However, these critical points when applying machine learning are not discussed in most of the works which use machine learning to build performance models. APARF is the only approach taking into account balanced and representative datasets for task-based parallelism.

Chapter 3

Background

In this chapter we present an overview of the framework in which this work has been developed and of multiple techniques that are applied in this thesis. First, performance analysis and performance tuning are described. Then, a global overview of machine learning and the algorithms used in this thesis to automatically generate performance models are disclosed.

3.1 Performance Analysis and Performance Tuning

When executing parallel applications, the main objective is to reduce the execution time in the same proportion as the resources increase. This objective in some cases is impossible to accomplish due to limitations, such as synchronization between resources, memory bandwidth, and other factors. Even though this goal may be impossible, developers expect to obtain the maximum possible speed-up while minimizing the performance problems that may occur in parallel applications. Therefore, performance analysis and tuning becomes an important task to reduce execution time of parallel applications.

Improving performance can be divided into three main tasks [37]:

- **Monitoring**. A representation of the behaviour of an application when executed is necessary before analysis can be performed. The behaviour is represented with metrics describing performance which can provide numerical values or graphical depictions, an example of metrics being execution time and the values of hardware performance counters. Additionally, performance monitoring is achieved using profiling tools or with the addition of code in an application (instrumentation) to collect the desired performance metrics.

- **Analysis**. The representation of the behaviour of an application obtained by means of monitoring must be analyzed to find possible problems, this task is called performance analysis. Performance analysis can be performed either manually or tools can be used to help interpret the values of metrics or to apply automatic analysis. An additional objective of this task is to analyze the origin of performance issues and find solutions which enable an improvement in performance.

- **Tuning**. After performance problems are analyzed and possible solutions are found, the next step is applying solutions which may improve performance. For performance tuning it is important to know where, what and when changes should be applied. Furthermore, modifications should be applied without inducing abnormalities in the application's behaviour, such as incorrect outputs or deadlocks.

There are two possible approaches when applying performance analysis and performance tuning. Each approach with their own benefits and drawbacks [38]:

- **Static**. The application is executed and a post mortem analysis is performed. The main benefits are that a wide range of modifications can be applied and more complex analysis can be performed. The main drawback is that only future executions may have their performance increased.

- **Dynamic**. Performance analysis is performed while the application is in execution and changes may also be applied while in execution. The main benefit is that the current execution may have its performance increased. However, the drawbacks are that the analysis is limited because it is performed while in execution, and the modifiable parameters are limited to those which can be modified dinamically.

Taking into account the characteristics of both approaches, the best method in terms of performance improvement is static tuning if the applications shows a deterministic behaviour because more parameters can be modified. However, in terms of adaptability, dynamic tuning is better as it can adjust parameters at execution time, which is essential if an application's behaviour can change during its execution or even between executions.

As this thesis is mainly focused in dynamic tuning, an extended explanation of the dynamic approach is performed in the following subsection.

3.1.1 Dynamic performance tuning

In dynamic performance tuning, an application is executed with tools which allow to analyze its performance at runtime. After the analysis, if necessary, changes are applied to improve performance while the application is in execution.

Because both performance analysis and tuning must be applied on the fly, it is difficult or impossible for a user to do this approach in real-time. Therefore, both automatic analysis and tuning are necessary.

A general overview of dynamic performance tuning is shown in Figure 3.1. In this view a new component is found: the performance model. A performance model is necessary to accomplish both automatic analysis and tuning, as performance models consist on the fundamental logic to analyze performance and how to solve detected problems.

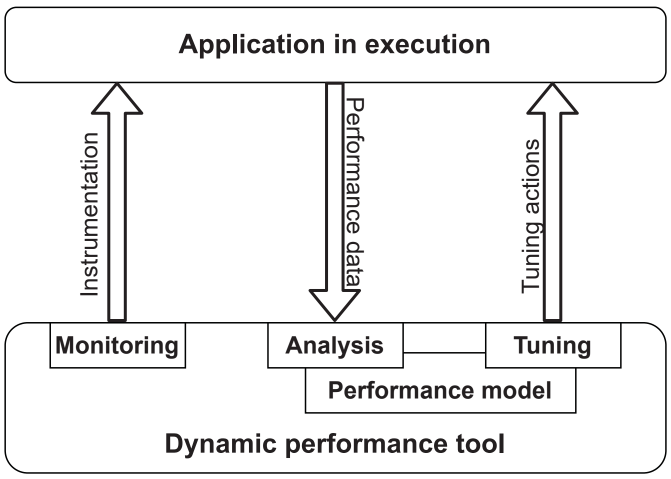


Figure 3.1: General overview of dynamic performance tuning.

First of all, the tool for dynamic performance tuning should specify which instrumentation and where it should be inserted in the application. Furthermore, a mechanism for the automatic insertion of instrumentation should be available, such as Dyninst [39]. Additionally, results provided by the instrumentation must be accessible by the tool.

During the application's execution, when the metrics values are available, the performance model is used to analyze the behaviour of the application and detect performance problems. If there are performance problems, the model must contain a methodology to provide adequate solutions to the tuning step. The performance model is built using previous knowledge about performance problems, allowing the model to automatically achieve a proper analysis and propose proper tuning strategies.

The last step in the process is tuning. Tuning is where the necessary changes proposed by the performance model are applied in the application to solve performance issues. Because in this case dynamic tuning is applied, it it important to find which parameters can be correctly modified in real-time and discard those which cannot be safely modified while the application is in execution. To achieve dynamic performance tuning, the proper parameters should be modified in a way that the execution consistency is preserved. The tuning step should include which parameters should be modified, how they are modified and when can such parameters be safely modified.

Performance model

The most critical points when applying dynamic performance tuning is using the proper performance model. In this subsection performance models are explained using an example model. This model is described in [11] which is built for OpenMP parallel applications.

First of all, the dynamical modifiable parameters for the application have to be identified, in the case of OpenMP parallel loops, these parameters are the number of threads and iteration's scheduling.

Then, the performance model which must correctly represent the behaviour of an application must be built. The model should take as inputs performance metrics obtained through instrumentation. The output of the instrumentation is used by the model to tune the application. An overview can be seen in Figure 3.2.

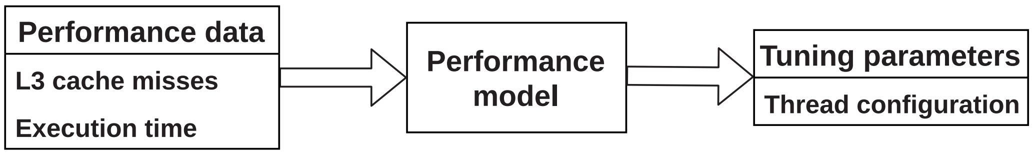


Figure 3.2: Overview of performance model.

The model described in [11] predicts the execution time for different threads configurations. This model is used to calculate the ideal number of threads with the minimum execution time, maximum efficiency or other possible combinations between execution time and resource usage.

The prediction is performed using execution time and the third cache level as inputs in the following way:

- Monitor L3 cache misses and execution time. L3 cache misses and a ratio between cache misses and time are considered to identify execution time for a particular number of cores in usage.
- Modify number of threads between 1 and the maximum number of cores in a processor. The monitoring is performed in one processor, where L3 misses and execution time are collected, starting from the serial execution and ending in the parallel execution with the maximum number of cores.
- Model relationship between metrics and predicted execution time for multiple processors. The collected metrics are used to build ratios between L3 cache and execution time. Then, a prediction of execution time from 1 thread to the maximum number of cores available in the system is performed.
- Extract ideal thread configuration from the model. This model predicts the execution time for all thread configurations and the user is responsible of providing the objective function to calculate the ideal number (such as maximum speed-up or efficiency).

Performance models can be built using the step performed by the example model: 1) selection of metrics and monitoring; 2) generation of different scenarios to monitor the behaviour of metrics in an application; 3) use mathematical expressions or algorithms to model the behaviour between metrics and the parameters to tune; 4) generate outputs with the tuning strategy.

3.1.2 Performance analysis and/or tuning tools

Multiple performance analysis tools, some including performance tuning capabilities, are explained in this subsection. A selection of some representative tools are presented as the work performed in this thesis could be implemented into them to improve applications' performance. However, there are several other tools such as Paradyn [40], Paraver[41], Scalasca [42] and Vampir [43], which are widely used in HPC environments. Although most of them do not offer auto-tuning capabilities.

MATE

MATE (Monitoring, Analysis and Tuning Environment) [44] [45] is a tool to dynamically monitor, analyzing and improve performance of parallel applications. The logic for performance analysis and tuning of MATE is included in Tunlets (which include performance models) to modify the parameters which can improve performance.

Because MATE is a dynamic performance tuning tool, changes in the application are performed while the application is in execution, which reduces the possible tunable parameters. As an example, in OpenMP v4.0 [46] applications, the modifiable parameters at run-time are limited to the number of threads and scheduling. Furthermore, the model must have an small overhead as both performance analysis and changes to the application must be applied as soon as possible to obtain the maximum possible performance improvement.

MATE has three main modules to analyze performance and apply modification in applications:

- The **Application Controller**, also called AC, is a program executed in each host the application is executed with. The AC is responsible of executing the application, it also modifies the application with the insertion of instrumentation to measure the appropriate metrics and modifies the tunable parameters of the application.
- The **Analyzer**. The main task of the Analyzer is, as it name suggests, to analyze the performance of an application using a predefined Tunlet. The instrumentation in the application sends values of the measured metrics to the Analyzer, which then applies the analysis step of the Tunlet. Then, if necessary, it provides to the AC instructions to properly tune the application.
- The **Dynamic Monitoring Library** (DMLib) is a common shared library for the ACs. DMLib is loaded into the application by the AC and it contains different snippets of code which can be inserted into the application. This library contains code which needs to be inserted in the application if required, such as instructions to monitor and modify application's parameters.

Performance models are described in the Tunlets. Each Tunlet is built to solve a performance problem and is composed by:

- **Measure points**. It describes what kind of instrumentation and where it should be inserted into the application during run-time.
- **Performance model**. Provides a methodology to analyze performance in an application with the objective of finding performance problems, which could be deteriorating the performance of the application. Moreover, it provides information about possible approaches or parameter's values to improve performance with the help of the next part of the Tunlet as described below.
- **Tuning points, action and synchronization**. In this part the modifications to the parameters are described with information about which changes and in which part of the application should be applied. Additionally, how the application should be synchronized when changes are applied into the involved processes to guarantee the consistency of the executed application.

A tool similar to MATE was developed to solve the inherent problems which may appear in MATE due to its centralized analysis model. In MATE, there can be multiple Application Controllers whereas there is only one Analyzer receiving and sending messages to all the ACs in execution when using distributed computing paradigms such as MPI. Consequently, one process (Analyzer) must receive performance data from multiple sources(ACs), apply performance models to analyze all the incoming data. Then, discover if there are performance problems to solve, and send to each Application Controller the appropriate tuning information. This task can be impossible to do at execution time in a timely manner depending in the required number of Application Controllers when executing an application.

This alternative tool is called ELASTIC [47] and defines a hierarchical way, in the shape of a tree, to evaluate performance with multiple analysis processes. The objective of this tool is to use multiple processes to balance the load of the single Analyzer between multiple nodes. This approach solves scalabity issues which appear in MATE when a distributed application requires the creation of a high number of Application Controllers.

DiscoPoP

DiscoPoP [48] is a tool for sequential applications conceived with the objective of finding regions of code where parallelism could be applied. Then, advice is provided to user as to how regions could be parallelized.

While the application is being executed, code regions which follow a pattern where memory is read, a computation is performed and memory is written, are identified as *computational units* (CU).

The different *computational units* identified in an application are used to build a dependency graph. This dependency graph is then used to detect data races and control operations. After the detection is performed, an analysis is performed to find *computational units* which can be executed concurrently with other CUs to be grouped as tasks. The dependency graph classifies regions into four candidates:

- **DOALL**. Simple loops where there are no data dependencies between all the iterations of the loop.
- **Reduction**. Loops where a computation is done to all the elements in one array and stored into a single variable.
- **Task parallelism**. Independent code regions which can be executed concurrently because there are no data dependencies between the different regions.
- **Pipeline**. Loops where only partial overlapping is possible in some consecutive iterations due to data dependencies.

After the classification is performed, OpenMP constructs are recommended to the user as the foundation for the OpenMP parallel implementation of each code region.

Periscope Tuning Framework

AutoTune's [24] european project used Periscope [49] (a performance analysis environment to help programmers analyze the performance of an application) to develop the Periscope Tuning Framework (PTF) [25] [50]. PTF is a tool for automatic performance analysis and tuning for both performance and efficiency.

PTF makes use of plugins to tune an application and each plugin is used to analyze different performance parameters, such as load balancing, energy consumption, data locality and memory accesses. The performance analysis can be applied with only one or multiple plugins to either explore one or multiple aspects in one application.

In PTF, the different code regions to analyze are executed with different variants, where each variant has different values for the parameters to tune. After the analysis, a report is generated with the values of the objective metrics (such as time and efficiency) for each individual code region and variant.

A plugin is divided into different steps which define the workflow to follow by PTF. The steps are:

- **Initialization**. Initialize the plugin's variables, tuning space and search algorithms for the tuning space.
- **Create scenarios**. The plugin explores the tuning space with the selected search algorithm and generates the different scenarios to be executed. This process may be repeated in the case additional scenarios are needed.

- **Prepare scenarios**. Executions for each scenario are prepared modifying the tunable parameters, either with the modification of execution parameters or recompiling the application if necessary.
- **Define experiments**. All the scenarios are assembled into one experiment and the different scenarios are executed. If different scenarios belong to individual code regions in the same application, multiple scenarios may be executed in one process. However, if the scenarios belong to the same code region, the application must be executed multiple times.
- **Restart information**. In this step the application is restarted and the necessary steps to apply the tuning actions are executed.
- **Process results**. If there are no more scenarios to be executed, the objectives of the plugin are analyzed, generating the performance report for the user.

PTF includes multiple plugins, such as plugins to tune compiler flags, energy, MPI parameters and number of workers in master/worker applications. Also, users can generate new plugins to meet their needs.

Kernel Tuning Toolkit

Kernel Tuning Toolkit (KTT) [6][51] is an autotuning tool for graphical processing units developed for both OpenCL and CUDA applications. KTT modifies kernel's parameters with preprocessor macros.

KTT's tuning workflow can be divided in four steps:

- **The code region** (kernel) must be implemented using tuning macros, as a way to easily adjust parameters which can improve performance for a kernel.
- **Generate a space of tuning parameters**. The space is generated with the parameters to adjust and a range of values for each parameter must be defined. There is the possibility to define constraint for the possible combinations of values.
- **Inputs and outputs**, which are assigned to kernels to test the correctness of the outputs generated by kernels after the tuning parameters are adjusted.
- **The tuning space is explored** with the selected search strategy.

Similarly to Periscope Tuning Framework, KTT explores the multiple parameter configurations. When tuning is finished, a report is generated with the combination of parameters which better improve performance.

Furthermore, KTT does not only provide local optimizations, it can also be used for global optimizations for multiple kernels with the same tuning parameters requirements.

TAU

The TAU parallel performance system [52] is a framework for instrumentation, measurement and analysis of parallel programs for HPC. Multiple performance tools and modules are integrated into TAU using interfaces and data formats so they can work together.

TAU's architecture is composed by three layers:

- **Instrumentation**. The first layer defines the metrics to be accessed in performance experiments. Metrics are used as event information and groups of events are generated if necessary when measurement is performed. Instrumentation can be inserted in multiple ways, such as manual annotation, with pre-processor instrumentation, compiler-based instrumentation, using wrappers, and additional approaches.

- **Measurement**. TAU supports parallel profiling and tracing to read the events declared in the instrumentation layer. The measurements for each experiment can be customized by users selecting the measuring modules to be used.

- **Analysis**. The results of profiling and tracing are reported. Users can analyze results either as a text or with the use of one or multiple graphical tools to visualize results.

The main strength of TAU is its flexibility as each layer can be configured by the user. Furthermore, there is a wide arrays of tools which have been integrated into it. This allows the user to select the tools they are more used to and also to select between a broad number of different metrics which may require the use of multiple frameworks.

Another advantage is that even though it was released almost two decades ago, TAU continues to be extended with new tools and capabilities as of today.

Additionally, TAU is able, with the help of SOSflow [53], to collect performance data dynamically [54] which opens the path for dynamic tuning in TAU.

3.2 Machine learning

Machine Learning (ML) is a branch of artificial intelligence. In this branch, models are built with the usage of algorithms, statistical models and sets of data with the objective of inferring patterns in data to generate predictions.

Machine learning approaches can be classified into two types according to the kind of data used [55]:

- **Supervised learning**. Each sample in the data has one or multiple labels which can be used as the output of the prediction. The label is used by the algorithm to infer relationships from samples with the same label.

- **Unsupervised learning**. In this case there are no labels in the data. The model should discover pattern from the data.

In the following subsection both approaches and some of their implementations are explained.

3.2.1 Unsupervised Learning

Unsupervised machine learning performs the training without knowing which are the data classes. The training step tries to find patterns and classify the inputs using patterns discovered or inferred by the model [56].

Some of the best known unsupervised learning methods are the following.

K-means

K-means is an unsupervised clustering algorithm. Given a pre-specified number of clusters K, the data is classified into one of the K clusters and each cluster has a point called centroid, which is the mean value of all the data points belonging to that cluster [57].

The steps when performing K-means are:

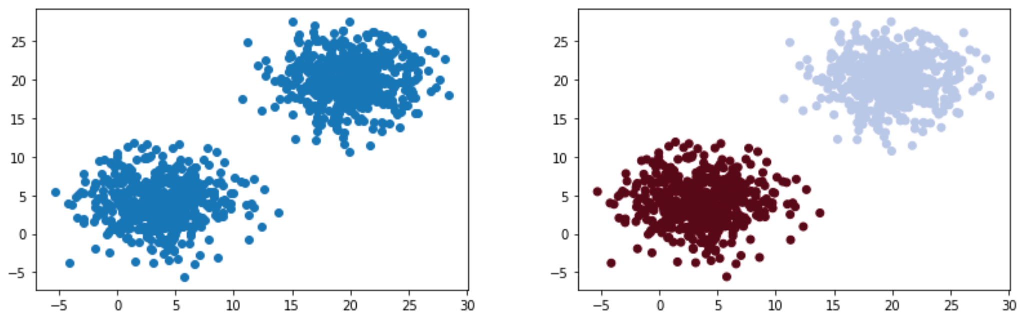
1. Set the number of clusters to K.
2. Initialize K centroids randomly from the data.
3. Assign each data point to the closest centroid.
4. Calculate the new position of the centroid with the mean of all data points assigned to it.
5. Go back to step 3 until the position of all centroids doesn't change or until a given number of iterations.

An example of a classification done with K-means can be seen in Figure 3.3. In Figure 3.3(a), the initial data is shown and two clear distributions of data can be seen as clouds of data points. Then, Figure 3.3(b) shows the result of applying K-means with $K = 2$ and, as expected, the two distributions of data are classified into the two clusters which could initially be seen.

Principal Component Analysis

Principal Component Analysis (PCA) is a method for dimensionality reduction. The objective of PCA is to use a set of data with a high number of variables (dimensions) and find a representation of this data reducing the number of dimensions. The new representation simplifies the number of variables without losing relevant information [58].

Each variable in the new representation is called a Principal component (PC), which are obtained though an orthogonal transformation of all the variables from the original data. Each principal component is linearly uncorrelated to the others and explains a percentage of the



(a) Initial data

(b) Example of two clusters found with K-means.

Figure 3.3: Example of K-means.

variability found in all the data. Each PC is assigned an ordinal number sorted from the one describing the highest variability to the one with the lowest variability. Because the components are sorted by variability, the dimensionality of the data can be reduced simply be removing the components which describe very low amounts of the variability [59].

Table 3.1: Example of variances found in each Principal Components

	PC1	**PC2**	**PC3**	**PC4**	**PC5**	**PC6**	**PC7**
Proportion	0,79	0,10	0,03	0,02	0,01	0,01	0,01
Cumulative	0,79	0,89	0,92	0,94	0,95	0,96	0,97
	PC8	**PC9**	**PC10**	**PC11**	**PC12**	**...**	**PC54**
Proportion	0,01	0,007	0,004	0,002	0,001	...	0,00
Cumulative	0,98	0,993	0,997	0,999	1	...	1

Let's use the example shown in Table 3.1. In this example we have as many principal component as variables the initial dataset had. The initial dataset is composed by 54 variables and principal component analysis is applied to obtain the new representation.

Along with the new representation, the principal components are generated and the first component (PC1) represents 79% of the total variance. With the cumulative variance, the number of PCs needed to represent a significant amount of the variance (90%) is only 3 components, although with only two a 89% can be represented. Furthermore, all the variance can be explained using 12 components. This is a big change compared to the initial dataset which had 54 variables and now with only 3 variables a high percentage of the dataset could be successfully represented visually in a three dimensional graph.

3.2.2 Supervised Learning

Supervised machine learning methods are used for datasets with labeled data. A label is a field on each entry of the dataset describing a property of the data which can be used as an identifier to classify each sample in the dataset. A dataset can have one or more different fields defined as labels (multi-label classification) [56].

Some of the best known and used supervised machine learning methods are explained below.

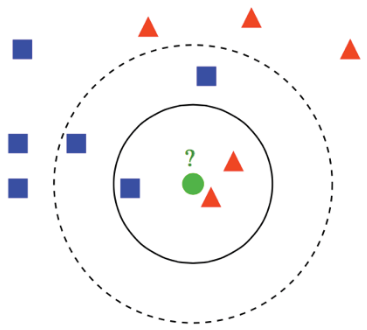
K-nearest Neighbor Classifier

K-nearest Neighbor Classifier is one of the simplest supervised methods in machine learning. Its simplicity is due to being non-parametric and only taking into account the raw input values it is provided in the training to classify. The classification uses both density and proximity of the training samples to the input data for prediction.

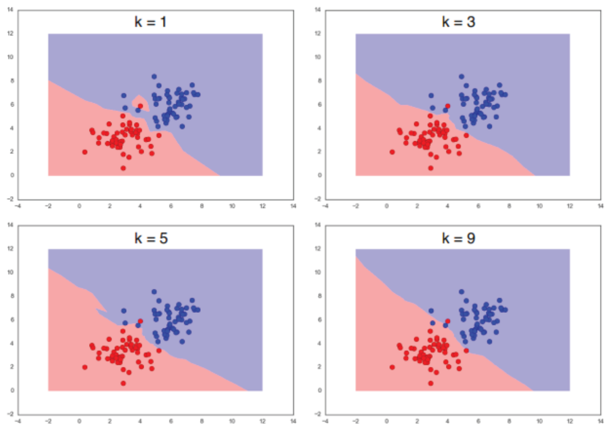
This methods is called *K-nearest neighbor* because the nearest K neighbors determine to which class a data point belongs to. In order to find which are the nearest neighbors for a prediction, distances (e.g. euclidean distance) are calculated between the data point to predict and all the data points the method was trained with. Then, the predicted class is the predominant label in the K nearest data points. Furthermore, there are variants of this algorithm where weights proportional to distance, instead of only density, are used in the prediction [60].

In Figure 3.4, from [61], two different examples for this machine learning method are shown. In the case of Figure 3.4(a), a green point is inserted into the space. Then, this point has to be classified as a red triangle or a blue square. If the value of K is 3, the space to check is defined by the space inside the inner circle, where two red triangles and one blue square can be seen. According to the density of triangle and squares, the number of triangles is higher, so with K=3 the green point will classified as a red triangle. In the case of K=5, defined by the outer circle, in the space of the circle there are two red triangles and three blue squares, so the new sample is classified as a blue square instead of a red triangle in this case.

In Figure 3.4(b), different examples of how the space is classified is shown for different values of K. According to the K parameter, the classification in some parts of the space changes due to class density.



(a) Example of KNN



(b) Example with different values of K

Figure 3.4: K-nearest Neighbor Classifier examples

The most important factor when using this method is choosing a correct value of K because the density inside the space of the chosen number of neighbors will decide the outcome. As the value of K increases, noise in predictions decreases but precision can be lost for similar classes with a low amount of samples in one region.

The main advantage of this method is its simplicity to implement. However, its accuracy depends on similarity between the elements of one class and how different that class is from others. Furthermore, this method is slow when performing predictions if a lot of samples are used in the training, as distances have to be calculated for each sample to predict against all the included samples.

Decision Tree

Decision Tree classifiers are directed acyclic graphs in the shape of a tree, where there is only one root node. An edge in a Decision Tree is a path between two nodes called father(ancestor) and son (descendant), nodes without descendants are called leafs. All nodes except the root and leafs are internal nodes [62].

Each node in a tree is a variable and the edges of each node represent the different decisions which can be taken. The output of each decision is another node and a leaf defines a predicted class.

In Figure 3.2.2 a simple Decision Tree can be seen. There is a computer which can overheat and there is an alarm to indicate whether the computer overheats or not. First, a check is performed to find out if the computer is on or off. In the case it is on, the temperature is monitored to see if there is overheating. The alarm only is enabled if a high temperature is detected.

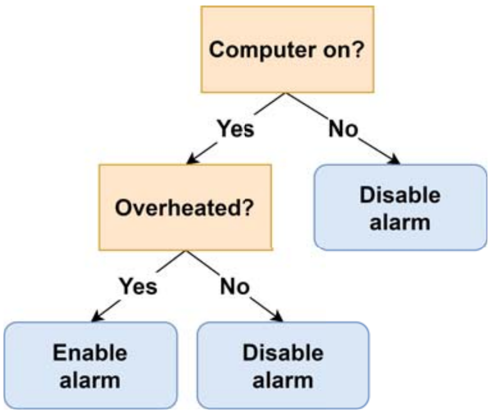


Figure 3.5: Example of decision tree classifier.

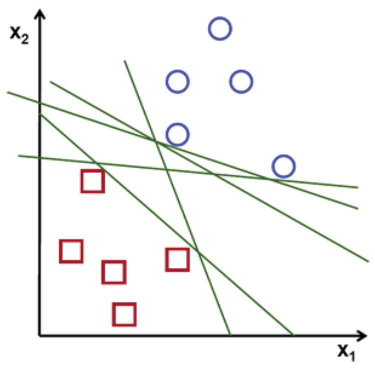
The main advantage of this method is the ease of reading and understanding how decisions are taken, as the architecture of the tree with its decisions can be visualized. One of the main drawbacks is the complexity to generate Decision Trees when there is a high number of variables and a high number of values for each variable.

Support Vector Machine

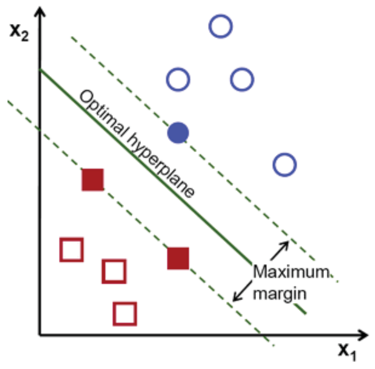
Support Vector Machine (SVM) classifies data using surfaces called hyperplanes which separate different labels of a dataset in different spaces.

Support Vector Machine finds a surface (hyperplane) which maximizes the space (margin) between the hyperplane and the nearest point of each class. This maximization has proven to reduce the generalization error [63]. The lines formed by the margins are called support vectors. In the case of linearly separable problems, the hyperplane can be written as: $w \cdot d + b = 0$, where d is the data to classify, w a vector and b a constant.

Figure 3.6 (a) shows a two dimensional space with two different labels (squares and circles) to classify. Likewise, there are some green lines drawn which define some of the infinite possible hyperplanes which separate the labels in two different spaces. The optimum hyperplane for SVM can be seen in Figure 3.6 (b), where the distance between the hyperplane and the nearest element for each label is maximized.



(a)



(b)

Figure 3.6: Support Vector Machine.

The main advantage of SVM is its computation speed once trained, which only requires to find in which side of the hyperplane the predictions falls into. However, it has some big disadvantages, such as not being suitable for big datasets due its training not being efficient in terms of computational power.

Artificial Neural Networks

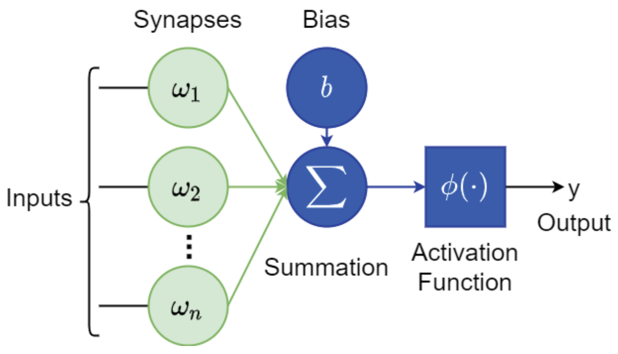
Artificial Neural Networks (ANN) is a method composed of a network topography. The topography is composed of one or more components called *processing units* or *neurons* and the connections between themselves (*synapses*). ANNs try to model the behaviour of the human brain[64].

A neuron is the fundamental unit of neural networks. Figure 3.7(a) shows how a neuron is modeled in ANNs. Neurons are located in both hidden and output layers of the artificial neural network.

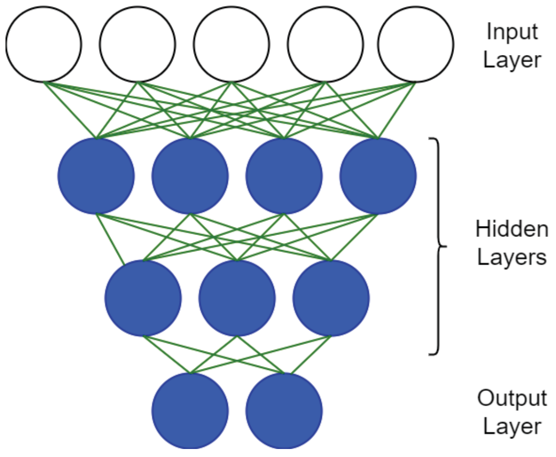
A neuron has three basic elements:

- **Synapses**. A synapse is a connecting link which has a weight. This weight is multiplied by the input of the synapse and the result is fed to the neuron. The weight is not limited to positive values.
- **Summation**. The weighted inputs are used as signals and a summation is performed using a linear combiner.
- **Activation function**. This function is used to limit the output's amplitude and normalize the output values of a neuron.

Some neural models include an additional parameter called bias. The bias is a constant real value added to the summation of the inputs before applying the activation function, allowing to shift the resulting values of an activation function.



(a) Model of a neuron



(b) ANN with two hidden layers

Figure 3.7: Example of a neuron and an artificial neural network. Parts of a neuron are shown in two colors in the artificial neural network.

The activation function can be linear or non-linear. In the case of using linear activation functions, the learning is limited to linear relationships between the data. However, most neural networks use non-linear activation functions [65][66] which allow the learning of polynomials with more than one degree. The are multiple activation function which are used in ANNs, most of them non-linear to ensure finding complex and non-linear relationships in the training data.

The architecture of artificial neural networks, as shown in Figure 3.7(b), is mainly composed by three types of layers:

- **Input layer**. First layer where the network takes input values and there is an element for each input.

- **Output layer**. Last layer where the output is obtained. One or multiple outputs can be defined by a model.
- **Hidden layer**. Middle layer, or layers, where both inputs and outputs are from and to other layers in the network.

When developing ANNs, there are multiple parameters which need to be adjusted, such as: learning rate, number of learning epochs, loss algorithm, optimizer, batch size, neurons in each layer and number of layers. There is no clear guidelines of how these parameters should be defined when building models.

Some advantages of Artificial Neural Networks are the ability to model non-linear relationships with non-linear activation functions. Furthermore, results can be generated fast as neurons in a layer can be executed in parallel. However, some of the disadvantages are the difficulty to understand the generated models and their training times, which are proportional to the size of the network.

3.3 Correlation analysis

Correlation analysis is a statistical method used to find the strength of association between two variables. Variables in correlation analysis are defined as vectors.

The three main methods for correlation analysis are **Pearson**, **Spearman** and **Kendall's Tau** [67]. These methods are explained as they have different meanings and can be applied to explain different scenarios.

Pearson is a parametric method to measure the strength and direction of linear relationships between two variables. The main problem with this approach being the assumption that data is both continuous and normally distributed. The correlation coefficient (r) is based on covariance and standard deviation between the two vectors. Eq. 3.1 is used to calculate the coefficient r, where x_i are the values of one vector and y_i values from the second vector, $\overline{x}$ is the mean of the first vector and $\overline{y}$ the mean for the second vector.

$$r = \frac{\sum(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum(x_i - \overline{x})^2 \sum(y_i - \overline{y})^2}} \tag{3.1}$$

Spearman is a non-parametric method to asses how an arbitrary monotonic function can describe the correlation between two variables. In contrast to Pearson, this approach makes no assumption about the distribution of the data and can describe non-linear relationships.

Spearman requires the execution of different steps to obtain the correlation coefficient. The steps performed to apply Spearman's correlation are:

1. Convert values to ranks. The list is converted to ordinals from the bigger value to the

lower. In the case of a tie (the same value in multiple positions of the vector), the mean of the ranks involved is calculated and assigned to them. As an example, given four values [0, 1, 1, 2] were two tied values appear, the ranks would be [4, 2.5, 2.5, 1].

2. Calculate the difference between each pair of ranks to compare.
3. Apply correlation formula (Eq. 3.2). Where n is the size of the vector and d the distance between each element in the same position of the two vectors.

$$r = 1 - \frac{6 \sum d_i^2}{n(n^2 - 1)} \tag{3.2}$$

Kendall's Tau is another non-parametric method to asses the correlation between two variables. In this case, the correlation coefficient represents concordant and discordant pairs between two ordinal variables. Even though this approach is for ordinal variables, variables can be transformed to ordinals as explained previously for Spearman.

The steps for Kendall's Tau are:

1. If not ordinal, convert values to ranks.
2. Sort vectors. The vector of the first variable is sorted from lower to higher. The second vector is modified applying the same position changes. For example, given vectors [1,4,3,2,] and [1,3,2,4], the first vector is sorted, obtaining [1,2,3,4]. Then, the same position changes should be applied to second vector, obtaining the "sorted" vector [1,4,2,3].
3. Find concordant and discordant pairs. After the first vector is sorted from lower to higher, the second vector should be checked to find if it is also sorted. For each element from the first element to the penultimate, a count is performed to find how many of the next elements in the vector follow the sorted distribution (concordant) and how many do not (discordant).
4. Apply correlation formula (Eq. 3.3).

$$r = \frac{concordant\_pairs - discordant\_pairs}{\binom{n}{2}} = \frac{concordant\_pairs - discordant\_pairs}{(n(n-1))/2} \tag{3.3}$$

An example with already sorted values can be seen in Table 3.2 to clarify how Spearman and Kendall Tau's work. In Spearman the difference between each value in the vectors is calculated and then Eq. 3.2 is applied, obtaining a correlation of 0.84. In the case of Kendall's Tau, starting from the first value of *Vector B*, which is *2*, there are four values which are higher. Therefore, they are considered concordant as they are correctly sorted. However, there is one value lower than *2* so a discordant value is found. This process is repeated until the penultimate element, and Eq. 3.3 is applied. The correlation for Kendall Tau is 0.73.

Table 3.2: Example of Spearman and Kendall's Tau correlation analysis between sorted vectors.

		Spearman	**Kendall**	
Vector A	**Vector B**	**Difference**	**Concordant**	**Discordant**
1	2	1	4	1
2	1	1	4	0
3	3	0	3	0
4	4	0	2	0
5	6	1	0	1
6	5	1		
		r=0.84	**r=0.73**	

The explained methods for correlation analysis have an output in the range [-1,1]. This range has three well defined scenarios:

- Correlation = -1. The two variables show a perfect but negative relationship (anticorrelation), when a variable increases the other decreases in the same proportion.
- Correlation = 0. There is no correlation between the two variables.
- Correlation = 1. The two variables show a perfect and positive relationship.

Chapter 4

Signature Reduction

This chapter explains how parallel code regions in an OpenMP application can be characterized. The behaviour is characterized using a signature composed by values of hardware performance counters. Additionally, the approach is aimed towards dynamic tuning, where metrics must be collected at execution time. Therefore, measuring values for all the hardware performance counters can be costly due overhead.

We explain how an initial list using a large set of hardware performance counters can be reduced systematically, so less performance counters are needed to be measured. This reduction lessens the measurement overhead, while still obtaining a valid representation of the execution of parallel code regions.

This chapter has two main hypothesis: (a) parallel regions can be characterized and identified at the processor level, using the values of the hardware performance counters available in the system and measured at execution time; (b) an equivalent but reduced list of hardware performance counters can be created to characterize and identify regions, while minimizing redundancy in the information obtained by the hardware counters.

First, a tool with the ability to read hardware performance counters dynamically is needed. Our research group developed MATE [44] [45], a tool for dynamic performance tuning. Because it lacked the ability to read hardware performance counters, it was necessary to modify MATE and integrate an API to extend its functionalities.

Performance Application Programming Interface [2] (PAPI) provides access to hardware performance monitoring with the use of a high level interface with high abstraction and a low level interface allowing more control to the user. PAPI with its low level interface was selected as it is one of the mainstream APIs for both tool integration and also to directly measure performance in applications. Additionally, PAPI allows the access of two different kinds of metrics which are called events. Events in PAPI are divided into native and preset.

Native events are the hardware performance counters available in the system, while preset metrics are common metrics the developers of PAPI consider to be of general interest in performance analysis. Among preset metrics we can find metrics such as different interactions in the memory hierarchy, cycles, instructions and operations. Furthermore, some metrics may not be present as hardware performance counters but may instead be able to be inferred from the available metrics, such metrics are also included in preset metrics as derived events. As an example, a system may have the number of total and conditional branch instructions but unconditional branches are not present in any hardware counter, in this case unconditional branches may be inferred subtracting the conditional from the total.

PAPI was integrated in MATE [68] and the performance model developed in [11], which was explained in Chapter 2.1.1, was used to verify its successful integration.

As explained previously, PAPI includes *derived* events, which are obtained applying different operations to multiple hardware performance counters. Derived events calculated using subtractions are the perfect candidates to be removed because some processors have synchronization problems when using PAPI with multi-threading. Therefore, it is possible to obtain events with negative values, a behaviour to avoid as hardware performance counters with negative values are meaningless.

In [69] we propose a methodology based on correlation analysis to determine the minimum set of hardware performance counters necessary to generate signatures of OpenMP parallel regions. The methodology is divided in the three steps shown in Figure 4.1. These steps are:

1. **Hardware performance data collection**. Execution data describing the values of hardware performance counters is collected and stored in a database. The execution data will be analyzed in a future step.
2. **Data exploration**. Principal component analysis is used to visualize the different kernels and verify whether they can be classified visually.
3. **Hardware performance counter reduction**. The most important step is to apply correlation analysis. Variables (hardware performance counters) are discarded if the correlation value between two, or more variables, is very high. Very high correlation points to the possibility of variables explaining similar information.

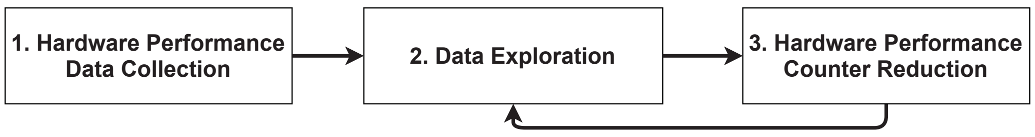

1. Hardware Performance Data Collection
2. Data Exploration
3. Hardware Performance Counter Reduction

Figure 4.1: Methodology for hardware performance counter reduction.

If the methodology successfully eliminates multiple variables, the redundancy in the dataset can be highly reduced which at the same time can help to:

- **Higher hardware counter measuring precision**. There is a limitation in the number of registers in processors to read hardware performance counters at the same time. Additionally, there are incompatibilities when accessing hardware performance counters, so some counters cannot be read at the same time as others. Because of this limitation, groups are created and compatible events are measured simultaneously in time intervals using multiplexing. If the number of events to measure is reduced, each group has more measuring time, increasing their precision.

- **Improved learning accuracy and reduced overfitting potential** [70] [71]. In machine learning the number of features is important. If there is a low number of features and/or the information is not relevant, a good model cannot be generated due to a lack of information. However, if there are too many features, where some of them can have redundant information, the model may overfit due to the irrelevant relationships in the data.
- **Lower computational cost**. With less events to measure, the overhead of obtaining the data is lower as less management is required. Additionally, with less metrics models with lower complexity can be used, reducing the time required for performance analysis.

In the following sections, the methodology applied to reduce the necessary hardware performance counters, without loss of performance information, is described in detail.

4.1 Hardware Performance Data Collection

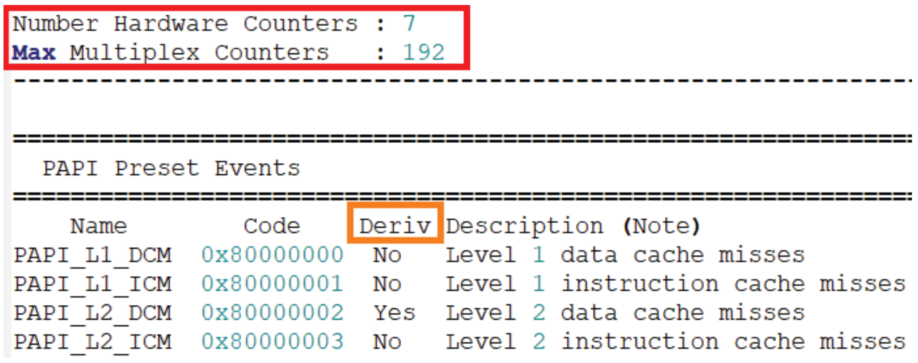
The initial step of the methodology is the collection of hardware performance data in a particular machine. This step should be done for each system with different architecture, because changes in the architecture (memory sizes, bandwidth, clock speed and changes in other components) modify the values and relationships between hardware performance counters, even if the same counters are available.

To limit the performance events to measure, we decided to only use PAPI's preset events, as they are a collection of common events relevant in application performance. In this list there are multiple events for branches, caches, cycles, instructions, TLB, loads and stores.

To obtain the list of available events, the command *papi_avail -a* should be used, which shows the results for only the available present events with the help of *-a*, which discards the unavailable events. Figure 4.2 shows a summary of *papi_avail*'s output, where three elements are highlighted:

- **Number Hardware Counters**. Determines the number of hardware performance counters which can be measured at the same time. This number is only for events that are compatible.
- **Max Multiplex Counters**. The maximum number of hardware counters which can be used with multiplexing.

- **Deriv**. An indicator for each event which reports if a particular event is derived (the combination of different hardware counters).


```
Number Hardware Counters : 7
Max Multiplex Counters   : 192
--------------------------------------------------------------

==============================================================
  PAPI Preset Events
==============================================================
    Name        Code       Deriv Description (Note)
PAPI_L1_DCM  0x80000000  No    Level 1 data cache misses
PAPI_L1_ICM  0x80000001  No    Level 1 instruction cache misses
PAPI_L2_DCM  0x80000002  Yes   Level 2 data cache misses
PAPI_L2_ICM  0x80000003  No    Level 2 instruction cache misses
```

Figure 4.2: Example output of papi_avail.

Once the list of available events is obtained, the next step is to check the compatibility between the different hardware counters. This step is done with the command *papi_event_chooser PRESET [EVENT]*, that reports which events are compatible with an event or a list of events, with the same output format as *papi_avail* (Figure 4.2). In the case of selecting incompatible events, the following error is printed: *Event [EVENT] can't be counted with others -1.*

After the list with all the events to measure is generated, a set of code templates must be executed to obtain execution information, allowing the analysis of the relationships between the different events. The code templates represent different simple OpenMP parallel region structures.

Each group of compatible hardware performance counters must be measured for each code template in different cases to check their similarity. Because some counters are highly dependant on the problem size, per example if the problem size is very small L2 and L3 caches may not be used. Therefore, it is necessary to measure the events with multiple problem sizes to obtain the realistic behaviour of the system. Furthermore, some code optimizations at the compilation level can impact the behaviour of the code, so more than one flag optimization value should be tested. For statistical significance and to discard possible outliers, multiple repetitions for each case are executed. Consequently, the collected data captures the behaviour for different code translations and memory patterns associated to the same OpenMP parallel region. The total number of executions for each regions is described using expression (4.1).

$$n\_executions = created\_groups * data\_sizes * flag\_combinations * repetitions \tag{4.1}$$

After the execution of all the different configurations, the data for each hardware performance group is joined in one dataset to be used in the following steps.

4.2 Data Exploration

The second step is the exploration of the obtained data. Principal Component Analysis (PCA) is used to visualize and validate the changes in the set of performance counters. This is possible because step 2 and step 3 are iterative and the changes applied in step 3 are validated in the next iteration of step 2.

First, PCA is applied to the data collected in step 1 and a new set of data is obtained. This new set is a representation where the execution information is transformed into principal components. This new representation allows us to check how much variance of the data is represented by each principal component. Furthermore, the variance described by each component can be used to determine the minimum dimensions needed for a visual characterization of the data with the minimum loss of information.

Thanks to the dimensionality reduction obtained applying PCA, the data can be plotted in 2D or 3D but incurring in some, but known, loss of information. With the 2D or 3D representation of the data, it could be possible to check if the executed templates can be visually distinguished from each other. Furthermore, if new executions were performed, we could identify the similarity between new executions and former executions with the differences in the PCA. Moreover, we could guess if methods such as clustering would be able to identify the data by region, compilation flag, problem size, or other possible classifications.

PCA can also be used to discover relationships between hardware counters because of the weights of each variable in each principal component. If two or more counters have the same weight, this coincidence could be due to having the same or highly related values, highlighting possible redundancies.

If the PCA resulting from removing a set of hardware performance counters and the previous PCA continue to be similar, and there is not a high impact in the variance explained with few dimensions, the reduction can be considered valid.

4.3 Signature Reduction

The main part of the methodology is the third step: Signature Reduction.

In this step correlation analysis is used to find pairs of hardware performance counters with high correlation coefficients. A high correlation between two metrics can indicate redundancy and one of the metrics involved may be removed, as a way to reduce over-fitting in machine learning algorithms caused by redundant features [72].

Accordingly, correlation analysis is performed over the dataset created previously in step 1. After the correlation analysis is performed, the obtained output is a grade of similarity between each pair of hardware counters. Similarity is described with values between 1 and -1, where

1 is highly correlated and likely redundant, 0 is no relationship between them and -1 a high anticorrelation. We expect that no negative correlations appear as it is counter intuitive that one hardware counter may decrease as another increases. Correlation values are obtained in a square matrix, where each row and column is an event, and the values in each field are the correlation between the different pairs of events.

However, looking only at the correlation analysis can be misleading. Since there are hardware performance counters which can have high correlation while not being redundant. Therefore, we should think if this correlation is because of redundancy or not. Given the following example code found in Listing 4.1, a strong relationship exists between the number of iterations (branches) and the number of memory accesses. This relationship is due to memory accesses being tied to *N*, which controls the number of iterations. In this case, and also in codes with a similar regular behaviour, the correlation between branches and memory accesses is probably near perfect. However, the counters have completely different meaning, so they should be kept as no redundancy is removed if one counter is discarded, but instead their removal generates loss of information. Nonetheless, in the case of the number of conditional and taken branches showing a perfect correlation, this relationship is logical and one of the events may be discarded.

```
int add(int N, float A, float B, float C)
{
    int i;
    for(i=0; i<N; i++)
        C[i] = A[i] + B[i]
}
```

Listing 4.1: Example code for correlation analysis.

The last part of this step, is to discard hardware performance counters. If two events are found to be highly correlated looking at their correlation coefficient and the relationship is logical, one of them is removed, generating a smaller signature. It is important to remove, if possible, derived events as explained at the beginning of this chapter. After an event is removed, step 2 (Data exploration) is performed again with the reduced dataset. In the event that there are no more redundant events found in the data, the current list of events is considered to be the smallest possible set without redundancy for the particular architecture.

4.4 Experimentation

In this section we explain the results obtained by applying the explained methodology. To validate the methodology a set of templates and two different processors are used. Also, to validate that the reduced list of hardware performance counters is able to create signatures of OpenMP parallel regions, while providing a valid characterization, a neural network has been trained to classify parallel code regions.

In this experimentation the templates used are different parallel code sections found in the

STREAM benchmark [73]. This benchmark has been slightly modified to separate each section in a code function for easier identification with performance tools. Furthermore, this benchmark has been selected as it can give an approximation of the behaviour found in real memory bound OpenMP applications.

This benchmark is divided into four different patterns, which are parallel regions of code, defined as (see Listing 4.2):

- COPY. All the elements of one vector are copied into another iteratively. There is one read and one store but no arithmetic operations are involved in the inner code.
- SCALE. Each element of a vector is multiplied by a fixed scalar value and the result is stored in a second vector. There is one arithmetic operation (multiplication), one read and one store.
- SUM. One element is extracted from the same position of two different vectors and the results of their addition is stored in a third vector. There is one arithmetic operation (addition), two reads and one store.
- TRIAD. This pattern combines the previous SUM and SCALE patterns into one. In this case, an element from vector c is multiplied by an scalar. Then, the result is then added to an element from another vector (vector b). At the end, the result is stored in vector a. There are two arithmetic operations (one multiplication and one addition), two reads and one store.

Two different systems have been used in the experimentation of this methodology, both with Intel processors. AMD was discarded as the hardware performance counters, in the systems at hand, do not allow the instrumentation of some memory cache levels.

On one hand, we have a DELL T7500 with two Xeon E5645 processors, with six cores per processor. Its memory hierarchy is composed by a 32KB L1 and 256KB L2 for each core, a shared 12MB l3 cache in each processor. The total amount of main memory is 96GB. On the other hand, there is a bigger machine, a DELL PowerEdge R820. In this system there are 4 processors, each processor with 8 available cores. The memory hierarchy is the same at L1 and L2 levels (a 32KB L1 and 256KB L2 for each hardware core), and a shared 16MB l3 cache in each processor. The total amount of main memory is 128GB. The summary for the two architectures is shown in Table 4.1.

4.4.1 Hardware performance data collection

The first step of our methodology dictates that the command *papi_avail -a* must be executed to obtain the present events available in the target processor. Then, determine the valid compatible groups for the available events. In the case of the Xeon E5645, PAPI reports 58 preset events available in the processor. The measurable events for this processor and the Xeon E5-4620

```c
void Copy()
{
    size_t j;
    #pragma omp parallel for
    for (j=0; j<STREAM_ARRAY_SIZE; j++)
        c[j] = a[j];
}

void Scale(STREAM_TYPE scalar)
{
    size_t j;
    #pragma omp parallel for
    for (j=0; j<STREAM_ARRAY_SIZE; j++)
    b[j] = scalar*c[j];
}

void Add()
{
    size_t j;
    #pragma omp parallel for
    for (j=0; j<STREAM_ARRAY_SIZE; j++)
        c[j] = a[j]+b[j];
}

void Triad(STREAM_TYPE scalar)
{
    size_t j;
    #pragma omp parallel for
    for (j=0; j<STREAM_ARRAY_SIZE; j++)
        a[j] = b[j]+scalar*c[j];
}
```

Listing 4.2: Parallel sections available in STREAM benchmark.

Table 4.1: Hardware used in the experimentation

	Dell T7500	**Dell PowerEdge R820**
Processor	Xeon E5645	Xeon E5-4620
# sockets	2	4
# cores per socket	6	8
Threads per core	2	2
L1 cache	32 KB	32 KB
L2 cache	256 KB	256 KB
L3 cache size	12 MB	16 MB
Main Memory	96 GB	128 GB

processor have been classified in the types as shown in Table 4.2.

Table 4.2: Events by type in the two experimented processors,

Type	**XEON E5645**	**Xeon E5-4620**
Branches	7	7
L1 cache	8	5
L2 cache	16	15
L3 cache	10	9
TLB	3	2
Cycles	3	3
Operations	3	3
Instructions	8	7
Total	58	51

Now, the four templates must be executed for the different configurations defined by: groups of compatible events, problem sizes and compilation flags. Additionally, for statistical significance and to take into consideration possible outliers, each configuration should be executed multiple times (repetitions).

Table 4.3 shows an overview of the number of configurations used in each system. The configurations are the same, with the exception of the groups of events necessary in each system to obtain all events. In this way, there are four templates which are executed for: 56 different problem sizes (ranging from an initial size of 3KB to 4.5GB); two compilation flags for code optimization (-O0 and -O2); the number of groups necessary to obtain all events (12 for Dell T7500 and 11 for Dell PowerEdge R820); and a thousand repetitions for each configuration. In total, there are a 448,000 entries with 58 columns (one column for each hardware counter) for Dell T7500. Consequently, according to expression 4.1: in Dell T750 for each template 1,344,00 executions are necessary, which is a total of 5,376,000 executions; in the case of Dell PowerEdge, this number is a bit less because there are 11 groups of events instead of 12, for each template 1,232,00 executions are necessary, which is a total of 4,928,000 executions.

After all the necessary executions' data is obtained and we can proceed to to step 2 (Data exploration) and apply PCA to the dataset with the values for all the different hardware per-

Table 4.3: Configurations to execute in each system

	Dell T7500	**Dell PowerEdge R820**
Processor	Xeon E5645	Xeon E5-4620
Templates	4	4
Groups of events	12	11
Problem sizes	56	56
Compilation flags	2	2
Repetitions	1000	1000
Total executions	5,376,000	4,928,000
Total entries	448,000	448,000

formance counters.

4.4.2 Iterative data exploration and signature reduction

Figure 4.4 shows the plotted PCA for the full set of events. This PCA shows that using only two dimensions, the data is appropriately explained, as around 89% of the data's variability can be visualized. Furthermore, the different previously explained templates from STREAM can be distinguished visually, even when only two dimensions are used. Additionally, using three dimensions further increases the differences between templates, so they can easily be classified. This initial result confirms the first hypothesis of this chapter: **parallel code regions can be classified using hardware performance events**.

An important information obtained from the PCA is the variance explained by each principal component. Table 4.4 shows the proportion of variance in each principal component, from the first principal component (PC1) until the accumulated variance is more than 99%, obtained with 11 dimensions. The total variance which can be obtained in plots, in the case of three dimensions, is 91.7% which we considered a good result as more than 90% of the variance can be shown visually.

Table 4.4: Percentage of variance explained by each principal component until 99% accumulated variance.

	PC1	**PC2**	**PC3**	**PC4**	**PC5**	**PC6**	**PC7**	**PC8**	**PC9**	**PC10**	**PC11**
Variance	79.17	9.968	2.588	2.002	1.209	1.072	0.861	0.757	0.576	0.516	0.499
Total	79.17	89.14	91.73	93.73	94.94	96.01	96.87	97.63	98.20	98.72	99.22

Table 4.5: Percentage of variance in each principal component after removing instruction cache events.

	PC1	**PC2**	**PC3**	**PC4**	**PC5**	**PC6**	**PC7**	**PC8**	**PC9**	**PC10**	**PC11**
Variance	80.68	8.6	3.455	2.865	1.598	1.002	0.771	0.544	0.359	0,08	0.026
Total	80.68	89.28	92.74	95.60	97.20	98.20	98.97	99.52	99.87	99.95	99.98

Another important information which can be extracted is the contribution (eigenvector) each variable has in each new dimension (principal component). One hint in finding redundant

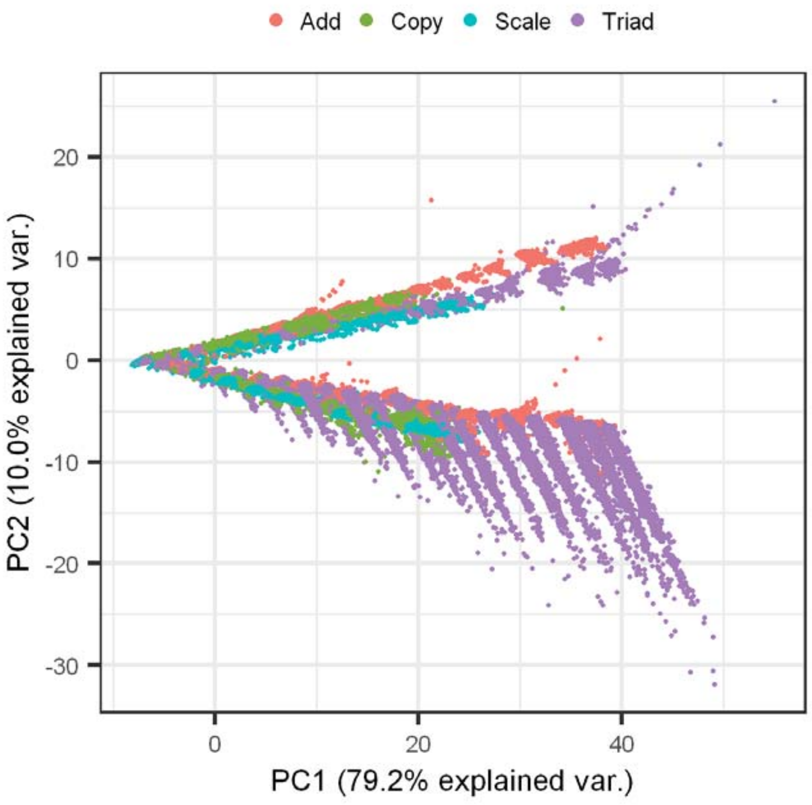


Figure 4.3: Initial results of applying and plotting PCA in two dimensions with RStudio.

variables is to find variables with the same contribution. This may mean that the two variables are equivalent, however it should be checked in the correlation analysis. An example, in the case of the XEON E5645, are the events *L3_TCW* (L3 total cache writes) and *L3_DCW* (L3 data cache writes) which are mapped to the same hardware counter (*L2_RQSTS:RFO_MISS* which is described by PAPI as "L2 requests, read for ownership misses").

The next step, *Step 3: Signature Reduction*, is to apply correlation analysis to the dataset to obtain the correlation value for each pair of events. The correlation matrix seen in Figure 4.4 is the result of applying linear correlation analysis to the initial dataset with all the hardware performance counters. In this figure, the darker the colour, the stronger the correlation between the values of one pair of events.

The absence of red (negative correlation) pairwise events is important. This absence is logical as all hardware events have increasing relationships between themselves, because the values of hardware performance counters remain the same or increase but cannot decrease. Using the correlation values we can analyze the strongest correlations in the matrix and decide which counters can be discarded.

Using the results obtained with PCA to get hints about hardware counter's significance, we realized that events related to the instructions cache, a total of 18, have low significance.

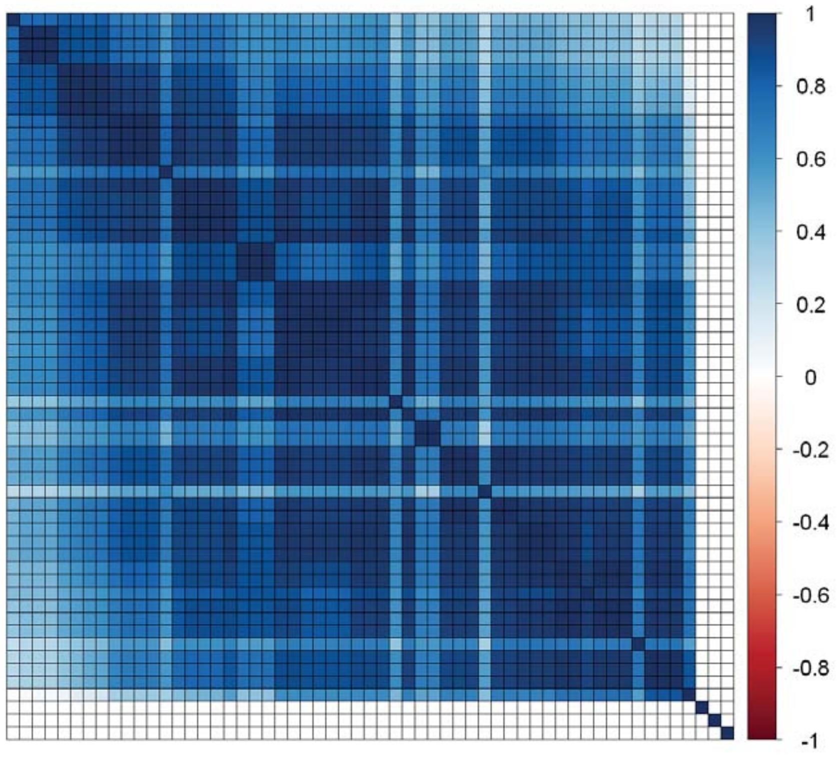


Figure 4.4: Initial correlation matrix with all the hardware performance counters.

This makes sense as they depend more on the code generated instead of the behaviour of the parallel regions. Additionally, when searching for performance models, we were unable to find any which made use of hardware counters related to the instruction cache. This allows to make a first reduction in the variables of the dataset from 58 columns to 40.

The new PCA after this simple reduction is shown in Figure 4.5. An easy to see change is the disappearance of some outliers in the new projection of the data. Now, in two dimensions the variance is around 89.3%, compared to the former 89% the difference is meager. The important change is, as can be seen in Table 4.5, that the values of variance for each principal component were updated. Before, with three principal components the accumulated variance was 91.7%, and now it has increased by 1%, reaching 92.7%. But the most important change is that the 99% of the variance can now be explained with only 8 components instead of 11, with higher variance explained in less components (99.52% compared to 99.22%).

As explained previously, a high correlation is not enough to discard a hardware event. There should be a logical reason by which the events are related, hence if it cannot be found, it is better not to use the correlation of the two events to remove either of them. As an example of logical correlation, there are two events which are very closely related with a correlation value of 100%, these are TOT_CYC and REF_CYC. As nowadays processors have technologies which

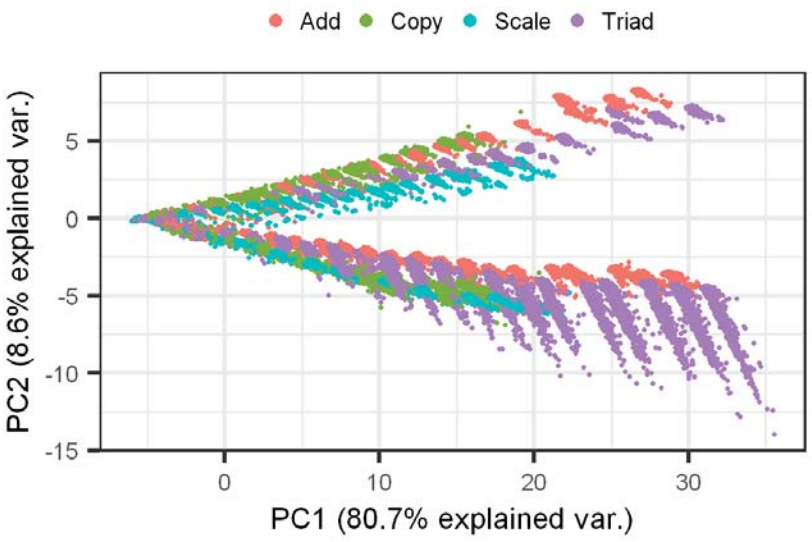


Figure 4.5: PCA in two dimensions after removing instruction cache related events.

allow them to change its frequency under certain circumstances, such as the resources in use, the temperature or the performance profile of the system, there is a dynamic frequency in the system. Due to the dynamic clock found in new systems, REF_CYC uses a reference clock as a way to obtain a fixed cycle representation which can be easily translated into execution time. Therefore, we decided to discard TOT_CYC and keep the reference cycles.

Moreover, we have discarded events which have access to the same resource, in this case we can find that for this particular machine the single point vectorization and double point vectorization operations read the same SIMD instruction's register.

Furthermore, with the correlation analysis we found some highly correlated events which are derived from the combination (addition or subtraction) of multiple events.

One clear case is related to branch instructions, as shown in Figure 4.6. Figure 4.6 (a) is the initial scheme of branch instructions available in the processor. In the figure, branch events can be shown as a tree because there is a clear hierarchy. Some branch events are a subdivision of another, so a missing branch event can be calculated if the other branch events are available. For example, if total and conditional branches are available, the unconditional branches can be obtained as the difference between the total and conditional branches.

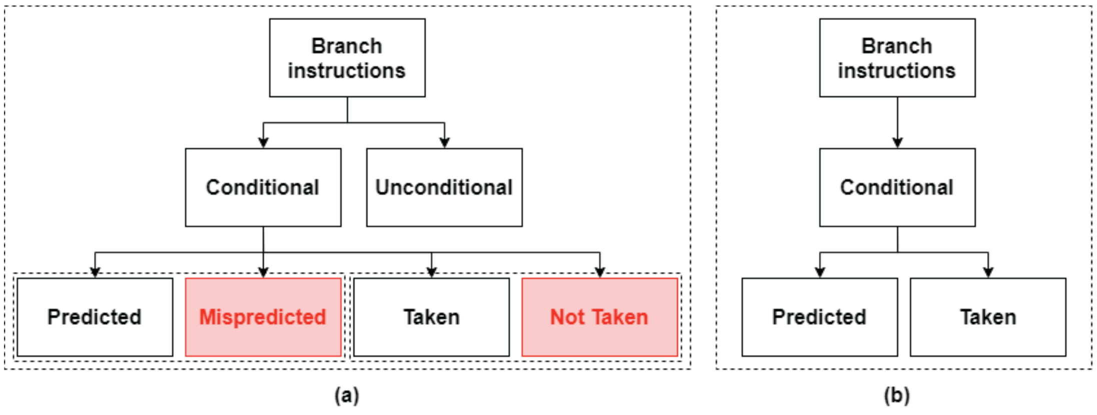


Figure 4.6: Initial and final scheme of branch instructions.

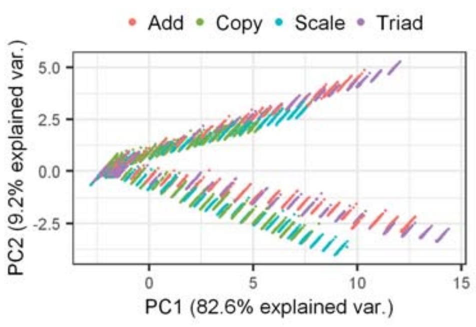
In this figure, the derived events calculated with subtraction are highlighted in red to indicate that they are the most appropriate candidates to be removed, because they are a combination of their father and one of their siblings.

Although the other events are not derived, they are theoretically the combination of the others branch events. The branch instructions are the addition of conditional and unconditional branches. In the case of conditional branches, there is a subdivision in two groups as indicated in the figure: correctly predicted and mispredicted, their addition being conditional branches; and the second group containing the taken and not taken branches, their addition also being the conditional.

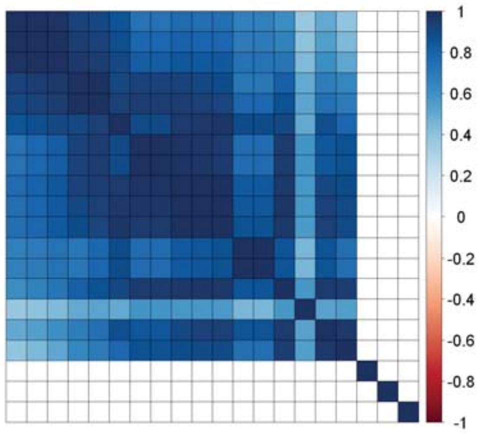
In the correlation analysis these events are among the highest correlated, with values close to 100%. Therefore, because of this high correlation and their logical relationships, some of them are removed. The final branch scheme can be seen in Figure 4.6 (b), the initial 7 branch events are reduced to 4, while no information is lost as the discarded events can be inferred from the remaining ones.

After iterating between steps 2 and 3 multiple times to analyze all the relationships while verifying the results, the list was reduced to 20 hardware performance counters. At the end, PCA is executed to verify the final list.

In the resulting PCA with the reduced list, which is shown in Figure 4.7 (a), the variance explained using only two dimensions is 91.8%. Compared to the initial 89.14 and the former 89.28 (removing instruction related events), there is a clear increase in the information explained with fewer dimensions. Table 4.6 shows the updated values for both variance per principal component and the accumulated variance. Now, with the reduced list of events, up to 96,12% of the information can be explained with only three dimensions, whereas in the initial and the former cases this value was lower than 93%. Moreover, with 4 principal components the total variance is more than 99%. This new result is important as in the previous cases, the 99% variance was only achieved with 11 and 8 components, respectively.



(a)



(b)

Figure 4.7: PCA and correlation matrix for the reduced list of events.

Table 4.6: Percentage of variance in each principal component for the reduced list of events.

	PC1	**PC2**	**PC3**	**PC4**	**PC5**	**PC6**	**PC7**	**PC8**	**PC9**
Variance	82,58	9,207	4,336	2,988	0,661	0,138	0,067	0,013	0,007
Total	82,58	91,79	96,12	99,11	99,77	99,91	99,98	99,99	100

Figure 4.7 (b) shows the correlation matrix for the remaining 20 events. Although there are still some dark points indicating strong relationship between some pairs of events, no logical relationship can be established among them, as a consequence, they cannot be discarded as information could be lost. As an example, L2 storage misses (L2_STM) shows high correlation to multiple branch events. However, there is no logical relationship between both, so they are kept. This high correlation could be related to the simplicity of the codes used which make use of the same simple loops and make the number of memory accesses directly proportional to the number of iterations.

Therefore, the reduced list cannot be further reduced and the minimum number of events, which characterize the behaviour of a particular node, is considered to be 20 hardware performance counters. In the case of the Xeon E5-4620, the list of events is similar, with 21 events. The result for both processors can be seen in Table 4.7, where events are classified in different types.

In the case of the second processor, multiple events were found to have negative values due to synchronization problems but they were derived events with subtraction. One example is L3 data cache accessed being the difference between L3 cache references and L2 instruction misses, which confirms that the perfect candidates for removal are redundant derived events based on subtraction.

Table 4.7: Events by type in the reduced list for the two experimented processors,

Type	XEON E5645	Xeon E5-4620
Branches	4	3
L1 cache	3	2
L2 cache	2	2
L3 cache	2	3
TLB	1	1
Cycles	1	2
Operations	3	3
Instructions	4	5
Total	20	21

4.4.3 Validation

The motivation behind the proposal of this methodology is the high number of available performance counters in current processors. This high number of counters makes their measurement at execution time costly and in some cases impractical. As as example to illustrate this claim, we use the templates explained previously and measure the initial list of events using multiplexing.

In cases were the problem size is small, for some templates the full list generates invalid values for the events. Sometimes the events have negative values, this is because the execution time is short and many groups of events are generated due to the incompatibilities. Therefore, some groups were not able to be correctly measured.

In contrast, when the reduced list of events was used, this problem did not appear in the tested templates. With the reduced list, we were able to asses the precision of multiplexing all the events and the introduced overhead. The overhead in the case of regions with execution time in the order of seconds, is up to 10 milliseconds and in the case of regions with lower time, the overhead is in up to 4 milliseconds. This overhead includes the time for both PAPI's initialization and instrumentation. As for the precision of multiplexing, in cases where execution time is less than a second, the general accuracy is between 90% and 99%, although unconditional branches are not properly estimated. For longer execution times (execution time higher than one second) the accuracy increases to more than 99%.

Finally we want to check if the two hypotheses stated at the beginning of the chapter are correct: (a) parallel regions can be characterized and identified using hardware performance counters; and (b) a reduced list of hardware performance counters, which reduces redundancy, can be obtained while fulfilling the previous hypothesis. The results obtained with the use of PCA seem to validate the hypothesis but we propose an experiment to add more evidence to this validation.

In this new experiment, a simple artificial neural network with only one hidden layer was trained using the dataset with the final list of hardware performance counter obtained by applying our methodology. The dataset is divided into two parts: a training set with the full dataset except two data sizes (432,000 entries); and a testing set with the two extracted data sizes (16,000 entries).

After the artificial neural network is trained for ten epochs, the resulting model is able to predict the two removed data sizes with an accuracy up to 99.98%. The results of this validation are relevant for two reasons: a) the reduced list of events can correctly represent parallel regions; and b) they hint that the signatures of parallel regions can be used in machine learning techniques.

4.5 Conclusions

In this chapter two hypothesis were taken into account: (a) identification and characterization of parallel regions can be performed using the hardware performance counters available in a system; and (b) the list of hardware performance counters can be reduced to minimize redundancy, while still able to characterize and identify parallel regions.

To prove both hypotheses, the monitoring of hardware performance counters is necessary and PAPI was integrated into MATE.

A methodology was created for the reduction of the list with three main steps:

- **Collect hardware performance data**. The values of hardware performance counters are collected with different kernels for multiple problem sizes and compilation flags.
- **Data exploration**. Principal component analysis is used to visualize the collected data from each kernel and obtain a visual representation. Additionally, PCA can help to find redundant hardware performance counters if they have the same weight in each principal component.
- **Signature reduction**. Correlation analysis is performed between pairs of hardware performance counters. If high correlation appears in a pair, the pair should be analyzed logically to check if the redundancy is real, so one counter can be discarded.

In the experimentation, PAPI preset events were used to generate a list of hardware performance counters for each processor, 58 events in Xeon E5645 and 51 in the case of Xeon E5-4620. A parallel OpenMP implementation for STREAM benchmark was used in the two systems to find redundant counters with 56 problem sizes and two compiler configurations.

After applying the proposed methodology, the list was reduced to 20 hardware performance counters for Xeon E5645 and 21 for Xeon E5-4620.

An artificial neural network was trained to check if the reduced list was able to correctly characterize the kernels. The generated model provided an accuracy of almost 100%, validating both the reduction and the two initial hypotheses successfully.

The methodology to generate an equivalent and reduced set of hardware performance counters was published in [74].

Chapter 5

Building Datasets for Performance Tuning

This chapter explains our proposal for building a representative and balanced dataset for performance tuning.

Results obtained in the previous chapter showed that a signature built with the values of the proper subset of hardware performance counters can be used to characterize the execution of an OpenMP parallel region for a given number of threads, binding and problem size. This means that it is possible to use these signatures for building a dataset of OpenMP parallel regions' executions with the objective of using it for performance tuning.

In the previous chapter we developed a methodology to significantly reduce the number of metrics due to redundancy. However, there is another challenge when creating datasets with signatures: how to determine if a given parallel region pattern shall be included in a dataset for training purposes?

Determining if a set of data should be integrated into a dataset or not is considered to be a crucial problem in machine learning and data mining. The accuracy of training a machine learning model with an imbalanced dataset is not representative of its global accuracy. When training a model, the imbalance generated by classes with a big skew in the number of samples provide more information for some classes compared to others, which may generate models unable to provide appropriate results when the target class is underrepresented [75] [76] [77]. Therefore, determining which parallel regions can be considered patterns representative of the behaviour found in other parallel regions, to generate representative and balanced datasets, is one of the most important challenges when building datasets for performance tuning.

The main objective of this chapter is presenting a methodology to generate a balanced and representative dataset of OpenMP parallel regions, which we call **pattern collection**. Each pattern included in the pattern collection should cover a unique portion of the N-dimensional space represented by the values of the hardware performance counters. Unique means here that each pattern should not fully overlap with other patterns in the search space represented by the dataset.

The methodology explained in this chapter makes the following assumption: the hardware performance counters used to generate code signatures have been already determined and reduced, using either the approach from the previous chapter (Chapter 4: Signature Reduction) or an alternative approach. Moreover, this methodology can be used either to expand properly built training sets or to create new datasets.

Figure 5.1 shows the general overview of the methodology proposed in this chapter. The current, potentially empty, pattern collection is assumed to be balanced and representative. We consider that a candidate covers a new part of the search space, and should be included in the collection as a new pattern, if it is not highly correlated with any pattern already included in the pattern collection.

In the case of finding an OpenMP parallel region, which may be an attractive candidate for inclusion in the collection, the following steps are performed by the proposed methodology to check if it covers a new part of the input space:

- **Characterization of a candidate kernel**. The candidate parallel region is executed to obtain its signatures. The execution is performed for different threads configuration in the system, in order to characterize all the possible combinations of threads and affinities in multiprocessor systems.
- **Building candidate kernel representation**. Once the signatures characterizing the candidate are obtained, the signatures are joined in a particular order creating a representation of the candidate kernel as a vector. This shape is necessary for further use in the correlation analysis.
- **Correlation analysis**. The last step is to perform a correlation analysis between the vector of the candidate and the vectors of each pattern considered to be representative, which are in the pattern collection, to determine the extent of their similarity.

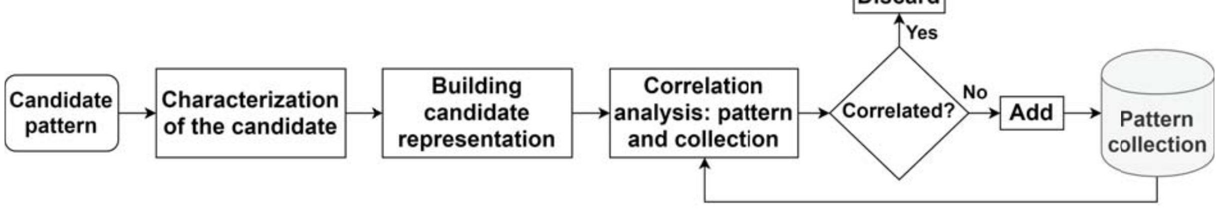

Candidate pattern
Characterization of the candidate
Building candidate representation
Correlation analysis: pattern and collection
Correlated?
Yes
Discard
No
Add
Pattern collection

Figure 5.1: Methodology to determine if a pattern covers a new part of the search space and should be included in the dataset.

After all the candidates are analyzed and the new patterns are included in the pattern collection, the collection is validated: a classifier model is trained with the pattern collection obtained after applying the proposed methodology; and the generated model is validated by classifying the discarded candidates, which should be classified as the pattern they showed more similarities to and lead to them being discarded.

5.1 Description of the pattern collection

The dataset built in this chapter is called **pattern collection**. But there is one main question: What does this dataset contain? We answer this question in this subsection.

First, from an outer perspective, there are all the executions for each pattern in the collection for a specific configuration. It is important to note that each dataset only works for a certain hardware configuration, so to characterize two machines with different hardware, a new dataset must be built, as the dataset probably will not work for the machine it was not created for. However, if there are multiples machines with the same characteristics, the dataset can be used for all of them. This limitation exists because for different hardware configurations we may collect different hardware counters values even if the same hardware counters are collected due to distinctive clocks in the systems, different memory sizes and other hardware characteristics of the processor and/or memory hierarchy.

Then, for each pattern in the collection the signatures for each combination of execution parameters are included, the execution parameters being: number of threads and their affinity, compilation flags, repetition number, problem size and/or other additional parameters if necessary. Furthermore, as this collection is used with the goal of, at a later time, build performance models, additional metrics and performance parameters can also be included as desired (ideal number of execution threads, execution time, scheduling policy, etc).

Furthermore, each pattern is executed for multiple problem sizes to have a representation of the behaviour in each memory level of the system's hierarchy.

5.2 Determining problem sizes for the pattern collection

This subsection introduces how we systematically determined problem sizes when building a pattern collection with the objective of stressing each particular memory level and generate a dataset that is also balanced with respect to the memory hierarchy.

When determining problem sizes, they must be directly proportional to the memory size in each level of the memory hierarchy. One the one hand, in the case of memory levels inside the processor (caches such as L1, L2 and L3), problem sizes must also be proportional to the number of physical cores in one processor. On the other hand, for memories outside the processor, problem sizes must be proportional to the number of processors in the system, instead of the number of cores.

Let's use Figure 5.2 as an example to explain the methodology to determine problem sizes for a particular system. The system in the figure is composed by two processors, as seen in Figure 5.2 (a), with three cache levels: private L1 per core, private L2 per core and a L3 which is shared between all cores inside the processor. In Figure 5.2 (b), the total memory by processor in each cache level is shown. In addition, there is the system's main memory which is shared between both processors.

Taking into account the characteristics of the system, the problem sizes are defined in the following way:

- **Private cache levels**. Each core in the processor has private memory to be used by the threads allocated in that particular core, in this case L1 and L2. To stress the multiple private cache levels, multiples problem sizes are defined which are directly proportional to the resources and defined by the multiplication of the number of private cache levels and the number of cores in one processor. Furthermore, for each cache level, problem sizes are defined starting with the size of one private cache and multiplied by the different core configurations, ending with the accumulated size of the private caches in the same level. In this way, in the machine presented in the figure, each L1 private cache has size 32KB and there are a total of six private caches, with a total of 192KB in L1. The resulting problem sizes in KB are: 32, 64 (Figure 5.2 (c) to (e)), 96, 128, 160, 192 (Figure 5.2 (f) to (h)). In the case of L2, each private cache is 256KB and the total is 1536KB, resulting in problem sizes (in KB) of: 256, 512, 768, 1024, 1280, 1536 (Figure 5.2 (i) to (k)).
- **Shared cache levels**. Each processor has some shared memory between all the cores and threads allocated in the processor, in the case of this system there is a shared L3 cache. In this case, the problem sizes to stress shared cache levels are bigger than the accumulated size of the lower level cache and slightly lower than the maximum shared memory in the current cache level. This range is used to ensure that the lower level caches cannot accommodate all the memory needed and, at the same time, the current cache level is not filled so the higher memory level in the hierarchy is avoided. With the characteristics of the example processor, six problem sizes are defined, one size per core in the processor. The first problem size with 2MB, which is slightly bigger than the accumulated 1.5MB of L2, and the last problem size of 11.5MB (slightly smaller than L3 cache of 12MB).
- **Main memory**. The biggest problem sizes are defined for the usage of the main memory. In this case the number of problem sizes is proportional to the number of processors in the system plus one. The initial problem size is bigger than the last level cache of the processor and the other sizes are obtained gradually increasing the necessary memory. The last problem size is 1.5 times the aggregated size of the last level cache of all the processors in the system. Using the machine from the example, where 12MB is the last level cache size in one processor, the problem sizes would be 12.5MB or slightly bigger for only one processor, for two processors 25MB and the latest problem size will be 1.5 of

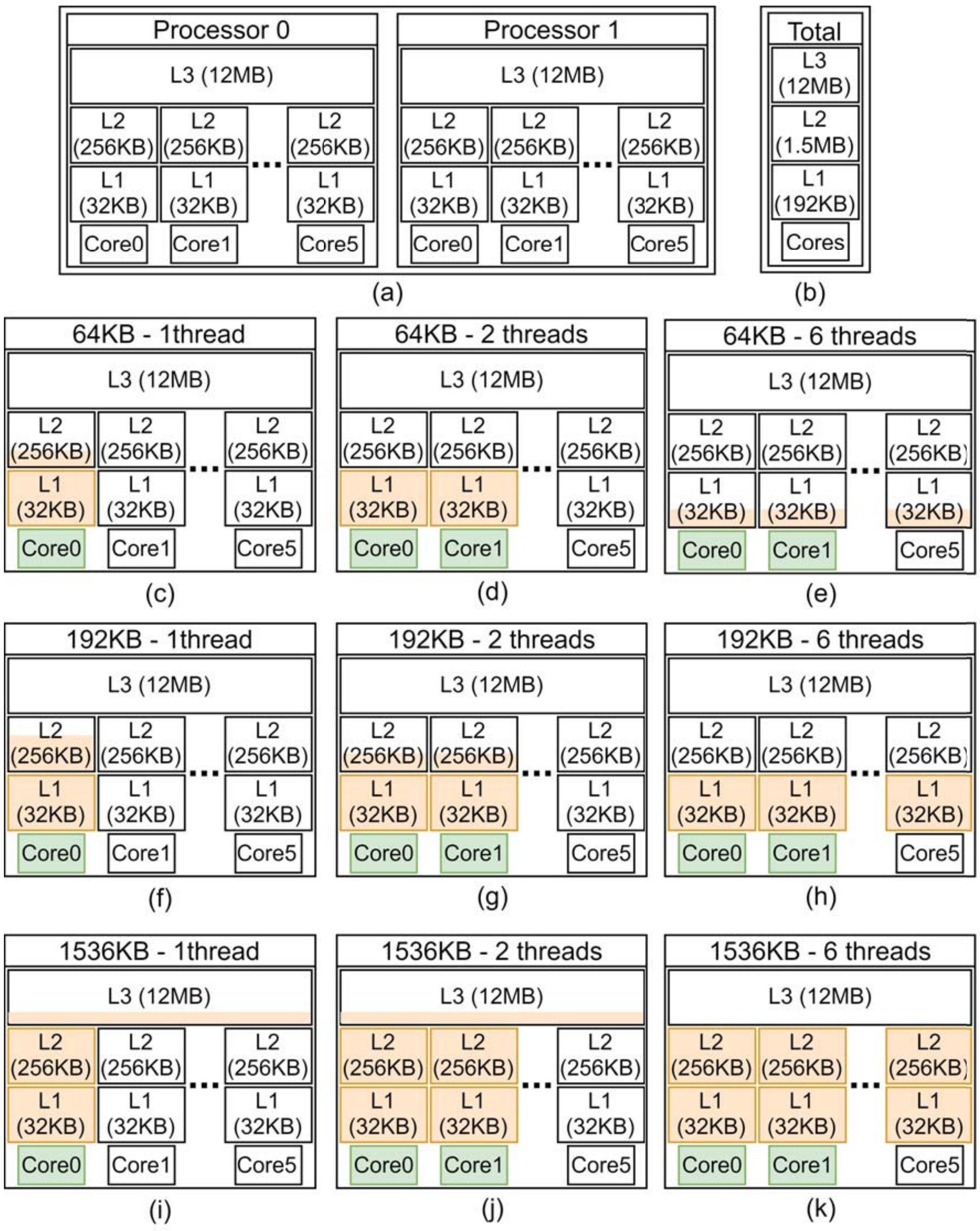


Figure 5.2: (a) Cache hierarchy and size in a system with two processors
(b) Total memory per cache level
(c-e) Cache usage with 64KB for one, two and 6 threads
(f-h) Cache usage with 192KB for one, two and 6 threads
(i-k) Cache usage with 1536KB for one, two and 6 threads

their aggregated capacity, which is around 37.5MB.

With this approach for defining problem sizes, two significant objectives are accomplished:

1. The problem sizes and their number is tailored to the memory hierarchy of the system the dataset it was created for, obtaining a dataset also balanced with respect to the memory hierarchy because the number of cases will be adequate to reflect its characteristics.

2. As the problem sizes are clearly defined, there is a better knowledge of the executions which stress each particular memory level. Enabling future refinements in the design of the dataset and for its use to train machine learning models.

5.3 Characterization of a candidate

The first step of the methodology is the generation of the candidate's signature to be assessed. In order to obtain the signature of an OpenMP parallel region in a given system, the region should be executed multiple times (for statistical significance) and also under certain conditions. The conditions for the executions are combinations of the following parameters:

- **Number of threads**. The parallel region is executed using different threads configuration available in the system, from the minimum number of cores (serial execution) to the maximum number of cores available in the system. As an example, if we have a machine with 12 physical cores, twelve configurations of this parameter are used from 1 thread to 12 threads, both included.

- **Thread affinity**. OpenMP offers two methods of assigning threads to cores, these are: close affinity, where threads are bound to contiguous cores; and spread affinity, where threads are bound in a round-robin fashion when there are multiple processors in the system. Given the impact of thread assignment in the memory footprint, specially in NUMA systems, the parallel region is executed using both options for every declared thread configuration.

Moreover, the problem size the candidate is executed with when obtaining the signatures for the correlation analysis is important. The overhead generated by OpenMP should be negligible, so a problem size which makes use of either L3 or main memory should be used.

Finally, with the objective of increasing the accuracy of the candidate's signature, hardware counter multiplexing is not used, so multiple executions are required to obtain all the counters' values, where each execution measures only one set of counters. Consequently, the total number of executions when executing a candidate kernels can be computed using expression (5.1)

$$n\_executions = repetitions * affinities * n\_cores * counter\_sets \tag{5.1}$$

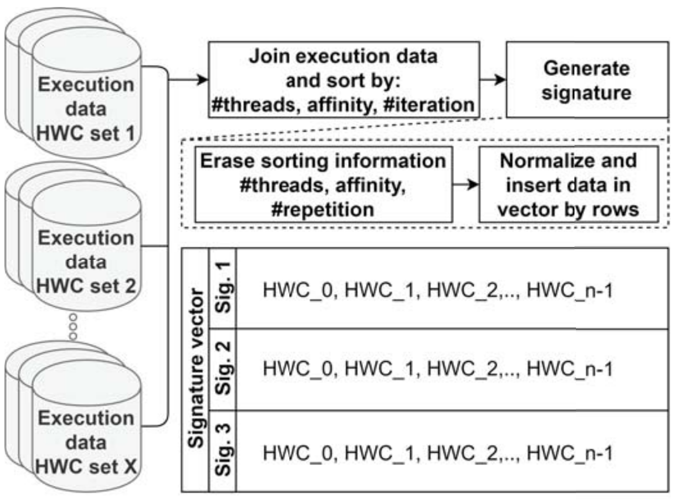

Execution data HWC set 1
Execution data HWC set 2
Execution data HWC set X
Join execution data and sort by: #threads, affinity, #iteration
Generate signature
Erase sorting information #threads, affinity, #repetition
Normalize and insert data in vector by rows
Signature vector
Sig. 1
Sig. 2
Sig. 3
HWC_0, HWC_1, HWC_2,.., HWC_n-1
HWC_0, HWC_1, HWC_2,.., HWC_n-1
HWC_0, HWC_1, HWC_2,.., HWC_n-1

Figure 5.3: Methodology to build the candidate kernel representation.

5.4 Building candidate kernel representation

The second step of the methodology can be seen in Figure 5.3. In this step, the representation of the candidate kernel is generated with the performance data previously collected.

The candidate kernel representation is built as follows:

1. **The execution data for each set of hardware performance counters is joined and sorted** by the number of threads used in the execution, their affinity and the repetition number. Additionally, as outliers can appear due to warming up cache, the one or two first repetitions for each case are removed.

2. **Remove non-hardware performance information**. The information used in the correlation analysis is related only to the signatures for each execution. However, as there is extra information such as the execution's configuration (threads, affinity and repetition number), hence this information should be removed from the dataset. This small step leaves only the information related to hardware performance counters describing the behaviour of the candidate.

3. **Normalization of the hardware performance values**. All the executions are normalized to avoid the values of the counters being directly proportional to the total execution time, obtaining a better behaviour representation of a candidate. This normalization is done with the counter related to cycles and each counter is divided by it, obtaining in each case a ratio of events per cycle. Another benefit of this normalization is that the values of the counters are between 0 and 1, except in cases where one event may be bigger than the number of cycles, which may happen in cases such as the total number of executed instructions.

4. **Flattening**. The dataset should be converted into a vector of signatures because correla-

tion is applied between vectors.

At the end, a sorted vector of signatures for all the executions, in the formerly specified conditions, representing the candidate is obtained and hence correlation analysis can be applied.

5.5 Correlation analysis

The last and most important step of the methodology consists of performing a correlation analysis between the patterns in the collection and the candidate. The correlation analysis is performed to determine whether the candidate covers a new part of the search space or not.

As it was mentioned in Section 3.3, Pearson's correlation is used in cases where data follows a normal distribution. As this assumption may not hold with signatures composed of hardware performance counters, this method has been discarded.

In this case the correlation analysis uses two different methods: Spearman and Kendall's Tau. Spearman's rank correlation is based on deviation between two series of data, and it is also more sensitive to data errors and discrepancies. On the other hand, Kendall's Tau is based on concordance and discordance between data pairs, which makes it effective for detecting trends. In general, result provided by both methods lead to the same inferences in the data, so the use of both methods serves to increase the robustness of our methodology.

The result from applying both correlation methods is used to decide whether a candidate should be included in the pattern collection or not, as its behaviour could already be represented in the space described by current pattern collection. If both methods indicate that the candidate kernel is not highly correlated to any pattern in the current pattern collection, this candidate should be included in the collection as a new pattern, because it can be considered to cover a new part of the search space.

The most important point when applying correlation analysis is determining a correlation coefficient threshold between the candidate kernel and the patterns in the collection. A threshold which can be regarded as high enough to ensure the exclusion of the candidate as a pattern. Threshold values for both methods may vary depending on the problem at hand, but a correlation coefficient higher than 0.7 is in general regarded as very high [78] [79].

Patterns that are not highly correlated would likely require different performance tuning strategies. Therefore, we adopt a stricter criterion and consider that the two patterns are highly correlated if their Spearman's rank correlation coefficient exceeds 0.9 (90%); and in the case of Kendall's tau, the correlation coefficient must exceed 0.8 (80%). Empirically, this combination also worked well for our problem as similar patterns reported correlation values which exceeded the defined thresholds.

Figure 5.4 shows an overview of the workflow performed in the correlation analysis. First, a correlation matrix is built and initialized to 0. This matrix will have one row for each pattern

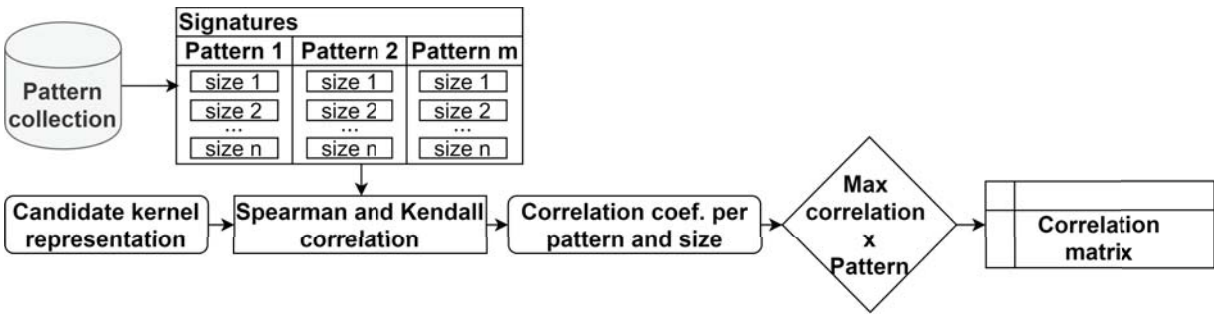

Pattern collection
Signatures
Pattern 1
Pattern 2
Pattern m
size 1
size 2
...
size n
Candidate kernel representation
Spearman and Kendall correlation
Correlation coef. per pattern and size
Max correlation x Pattern
Correlation matrix

Figure 5.4: Correlation analysis workflow.

in the collection and four columns to store the Spearman's rank, the Kendall's tau, and the P value of Spearman's rank with the statistical significance and the problem size.

Next, Kendall's and Spearman's correlation analysis are applied between the candidate kernel representation and all the problem sizes for each pattern in the pattern collection. The analysis provides both correlation coefficients and the Spearman's rank statistical significance. Then, for each pattern, the information associated with the maximum Spearman's rank (coefficient values, P value and the corresponding problem size) is stored in the correlation matrix.

Finally, the candidate kernel is discarded if it is highly correlated with at least one pattern in the collection. The criteria for discarding a pattern being at least one row in the correlation matrix with a Spearman's r>0.9, Kendall's tau>0.8 and a P value<0.05. If the criteria is fulfilled, the pattern is discarded because we consider that it does not cover a new portion of the search space. Otherwise, the candidate kernel is considered to cover a new portion of the search space and it is included into the collection as a new pattern.

To include a candidate as a new pattern in the collection, the candidate must be executed for all the problem sizes included in the pattern collection, in order to obtain the corresponding signatures.

5.6 Experimentation

In this section the proposed methodology is applied to incrementally construct a dataset of OpenMP parallel patterns. Additionally, the ANN described bellow is leveraged to validate the collection and demonstrate the importance of using a balanced dataset. Figure 5.5 summarizes the workflow followed in this experimentation:

- **Phase 1: Building the pattern collection**. The pattern collection is built applying the steps described in this chapter (characterization of a candidate, building candidate kernel representation and correlation analysis).
- **Phase 2: Validating the collection**. The pattern collection is validated and the importance of using a balanced dataset is illustrated using an ANN.

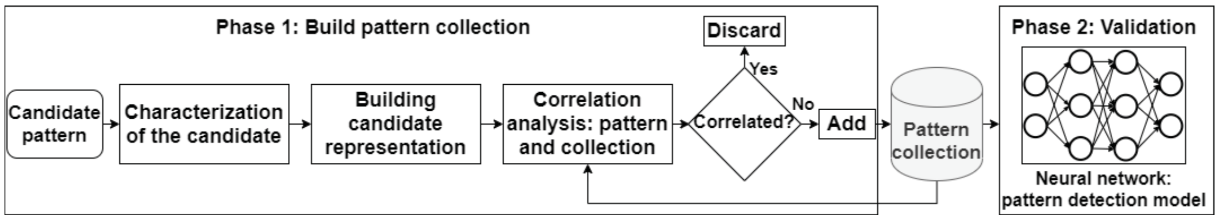

Phase 1: Build pattern collection
Candidate pattern
Characterization of the candidate
Building candidate representation
Correlation analysis: pattern and collection
Correlated?
Yes
No
Discard
Add
Pattern collection
Phase 2: Validation
Neural network: pattern detection model

Figure 5.5: Experimentation's workflow.

The candidate kernels for building the pattern collection are extracted from two different and well-known benchmarks:

- textbfSTREAM (Sustainable Memory Bandwidth in High Performance Computers) [80] is a synthetic benchmark composed of simple vector kernels to measure sustainable memory bandwidth. This benchmark was also used in the previous chapter.
- textbfPolyBench [81] [82] is a collection of benchmarks with multiple kernels. Version 4 was used which includes 23 different benchmarks divided in different categories (datamining, linear algebra, medley and stencils).

The experimentation for this methodology is performed in the same machines described in Chapter 4 (see Table 4.3 in Section 4.4.1). The main differences between the two machine is the number of cores of the system, as Dell T7500 has 12 cores between both processors while the PowerEdge has 32 cores between its four processors.

The explanation of the experimentation is performed using the Dell T7500. The results for the PowerEdge will be shown at the end of this section.

Both benchmarks were executed using float elements, so the number of counters used is less than the ones described in the former hardware counter reduction. This is because events regarding vectorization and doubles report values of 0, therefore these were also discarded to reduce the groups of events needed to measure, and to avoid having events with 0's in the dataset. Therefore, only 18 hardware counters are used as the features to create the executions' signatures.

To obtain the characterization of a kernel on Dell T7500, Equation 5.1 is used. According to the equation, 9000 executions are needed to generate 1800 signatures for each candidate. The following combinations of parameters must be executed:

- *Number of threads.* The machine has twelve cores, so executions from 1 core to the maximum of 12 are necessary.
- *Thread affinities.* The affinities available in OpenMP are close and spread as explained previously.
- *Number of repetitions.* 75 repetitions were executed to attain statistical significance.

- *Number of event sets.* To obtain all the values of the hardware performance counters five groups of events were necessary to cover the 18 counters.

Furthermore, when a candidate is regarded as a pattern, this number of executions (9000) is multiplied by the number of problem sizes needed to obtain the behaviour at the different memory levels. In the case of this system, 21 problem sizes are needed, so in total (9000*21) 189000 executions are necessary to fully characterize each pattern. If this number is divided by 5 (number of event sets), 37800 signatures are necessary to characterize a pattern in the collection.

Initially, the first group considered as possible candidates are the four kernels extracted from STREAM. These kernels were introduced in section 4.4 and their code shown in Listing 4.2

As the pattern collection is originally empty, the collection is initialized using the Copy pattern. Then, the methodology is applied for the remaining candidates. Therefore, as Copy must be included in the collection, the representation for each problem size must be obtained, and it is executed a total of 189000 times. In contrast, the remaining three kernels are candidates and must be executed only for one problem size, which should be big enough to minimize OpenMP's overheads for 9000 executions. MATE [44] [45] is used to acquire the hardware performance counters values for each of execution and compute the signatures for each kernel.

Once the pattern collection has been initialized with all Copy's signatures, and one significant signature has been computed for Add, Scale and Triad, we perform the correlation analysis. Table 5.1 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 5.1 also shows that there is a very strong correlation between the three candidates, which is clearly above the threshold. This relationship is observed in Figure 5.6, which shows that none of the candidate kernels are highly correlated with the Copy pattern, but also shows that they are strongly correlated to each other.

Table 5.1: Table with maximum correlation coefficients for STREAM kernels.

	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

As a result, only one of the candidates shall be included in the pattern collection. The chosen pattern is Triad because it is the result of the composition between Add and Scale, so it is logically the most general one between the candidates.

Consequently, after applying the methodology to the kernels extracted from STREAM, the

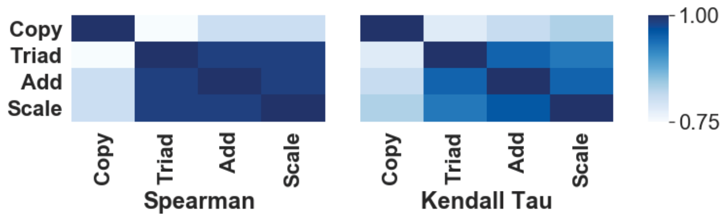


Figure 5.6: Figure with maximum correlation for STREAM kernels.

pattern collection will contain the following patters:

- **Copy**. Pattern abstracting memory accesses (read and/or writes) in consecutive memory positions.
- **One dimensional group**. The Triad's name is altered and renamed as a pattern which abstracts operations involving one-dimensional vectors.

Next, we extend the pattern collection using PolyBench from which we have extracted 29 new candidate kernels. PolyBench did not provide a parallel version, but we realized that it was simple to implement and OpenMP parallel version of several of the benchmarks stored in the directories *blas*, *kernels* and *stencils*.

After executing all the parallel kernels of PolyBench and generating their candidate kernel representations (signatures), we incrementally detected new patterns and included them into the pattern collection. These new detected patterns are described as follows:

- **Reduction**. Abstraction of reduction operations. Such as the addition of all the elements in a vector ($red = \sum c[i]$).
- **Stride**. Abstraction of non-contiguous memory access, memory accesses are performed with a certain stride ($c[stride \cdot i] = a[stride \cdot i]$).
- **Rows Stride**. Abstraction of memory accesses involving column-wise traversal of a matrix ($c[i \cdot N][j] = a[i \cdot N][j]$).
- **Matrix x Vector**. Abstraction of different matrix-vector operations, such as the matrix-vector product ($A = B \times v$).
- **Matrix x Matrix**. Abstraction of different matrix per matrix operations, such as the matrix-matrix multiplication ($C = A \times B$).
- **Stencil**. Abstraction of multi-dimensional stencil operations, such as: $A[i][j] = A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1]$.

Figure 5.7 shows the correlation analysis between the patterns included in the collection and PolyBench's discarded kernels. It can be clearly seen that all these kernels are highly correlated to at least one of the patterns in the collection.

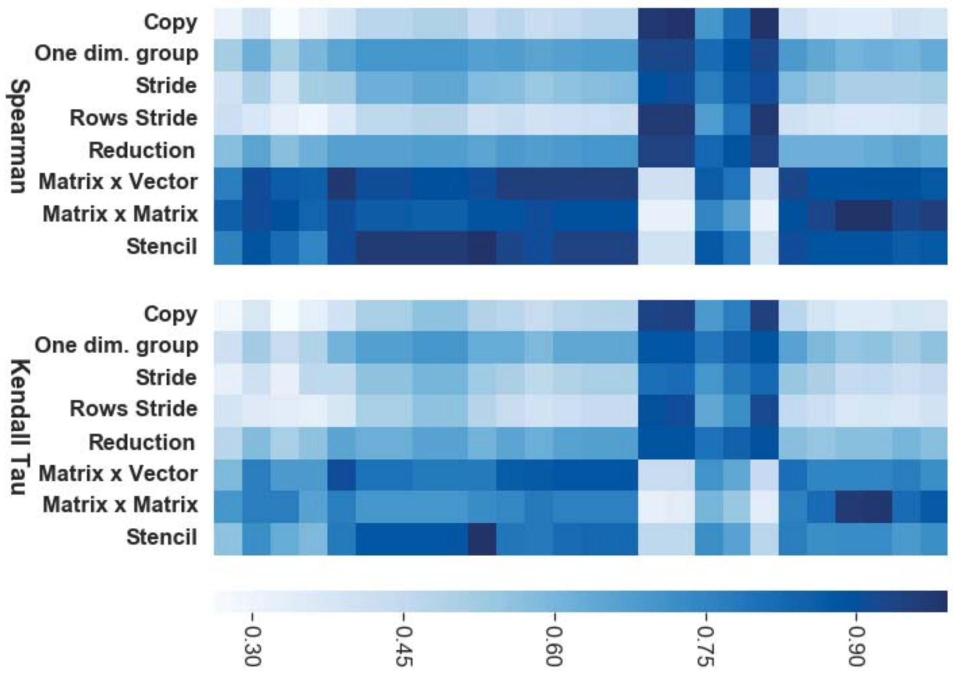


Figure 5.7: Spearman rank and Kendall's Tau correlation between PolyBench discarded kernels and the pattern collection.

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

The patterns obtained in the case of Dell Poweredge R820 with the same candidates are: **Copy, Scale, Triad, Stride, Reduction, Matrix x Matrix**.

The signatures of the discarded kernels were kept and will be used in the next section to validate the methodology.

5.7 Validation with pattern detection model

In order to validate both the methodology and the collection obtained in the previous section, we have used the pattern collection to train an Artificial Neural Network for generating a pattern classification model.

The ANN model is a Fully-connected and Feed-forward Neural Network with the architecture described in Table 5.2.

The first layer is the input layer where there are as many inputs as hardware performance counters in the kernels' signatures.

The SELU activation function was 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 [83].

The output layer utilizes the Softmax function paired with Categorical Cross-Entropy, allowing the model to perform classification for multiple classes. Therefore, the network outputs a vector of probabilities of a given instance in the dataset belonging to each pattern.

The potential for the network to overfit is counteracted with a probabilistic dropout and constraining the networks' weights [84]. During iterations of training, each neuron in the hidden layers of the network is temporarily removed with a 10% probability. Due to the low number of neurons in each network layer, increasing the dropout probability past 10% could prevent the model from converging. A constraint was applied to the weights incident to the hidden and output layers by clipping them to the range [-10, 10]. This helps to regularize the weights and prevents only a small number of them from dominating the network.

The Adam optimizer [85] was selected for training by stochastic gradient descent with learning rate $\alpha = 0.001$ and exponential decay rates $\beta_1 = 0.9$ amd $\beta_2 = 0.999$. The selection of these hyperparameters is less significant as the regularization provided by the SELUs allow for much higher learning rates and decays while maintaining a relatively smooth convergence rate [86].

As explained previously, each pattern in the collection was executed for 21 sizes, twelve thread combinations, two different affinities and 75 repetitions, so for each pattern there are of 37,800 signatures. Given that there are 8 pattern in the collection, the total number of signatures is 302,400.

The signatures have been divided in a 80% (241,920 signatures) for the training and a 20% (60,480 signatures) for the test. The artificial neural network has been trained for only 24 epochs using batches of 100 signatures. The model obtains a final loss of 0.0301 and an accuracy of 98.93%.

Next, a validation set has been built using the signatures of the discarded kernels from both STREAM and PolyBench. Additionally, multiple kernels have been extracted from the NAS parallel benchmarks (NPB) [87]. NPB is composed of 8 different benchmarks which are widely known and used in parallel supercomputers to study the performance of parallel systems [87].

Specifically, we used the following ten OpenMP parallel kernels extracted from the NPB:

Table 5.2: 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%

Table 5.3: Accuracy of the ANN for the discarded kernels.

	Number of kernels	**Accuracy**
Copy	3	0.93
1D group	3	0.9
Stride	3	0.91
Rows stride	0	-
Reduction	0	-
Matrix x Vector	5	0.9
Matrix x Matrix	5	0.87
Stencil	12	0.99

- *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*. Which are regions that have been extracted from different CG's functions.
- *l2norm_LU*. Which corresponds to the *l2norm* Lu's function.
- *ssor_LU*. Which is an OpenMP parallel region extracted from the LU's *ssor* function.

Table 5.3 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).

The results show that the discarded kernels in the classification model are generally classified correctly, obtaining an accuracy that ranges between 0.87 and 0.99. Although this is a result which verifies that the methodology works correctly, there are some cases where a more detailed explanation is needed.

For some of the discarded kernels, although the code looks similar to another pattern and the correlation analysis also defined it as the pattern the code is similar to, the accuracy is lower. One clear example is the parallel region *doitgen_1* extracted from the PolyBench benchmark *doitgen*. The code of this kernel can be seen in Listing 5.1 and it is clearly a *Copy*, but the model only detected 70% of the cases as *Copy*. Looking for the reason of this low accuracy, we saw that this particular region is one where due to restrictions in the code of PolyBench, the size of *sum* is small (defyned by *_PB_NP as 160*) and therefore the overhead caused by OpenMP is high.

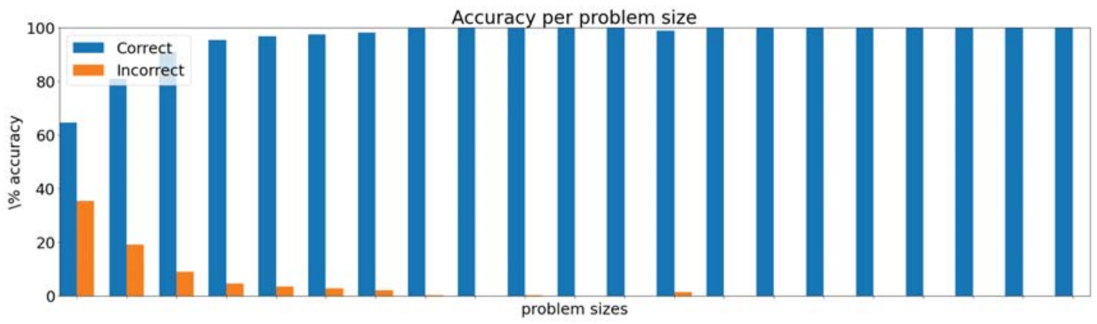


Figure 5.8: Accuracy for the ANN from lowest (left) problem size to the highest (right).

```
#pragma omp parallel for
for (p = 0; p < _PB_NP; p++)
    A[r][q][p] = sum[p];
```

Listing 5.1: Code of the doitgen_1 kernel extracted from PolyBench.

Moreover, this issue is also discovered when looking at the accuracy of the ANN model for each problem size. Figure 5.8 shows the accuracy from the smallest problem size (first from left hand side) to the biggest problem size (first from right hand side). It is clear that as the problem size increases the accuracy of the model increases too as the kernels have less overhead. Furthermore, the biggest problem sizes make use of memories in higher levels of the hierarchy, obtaining more information from hardware performance counters.

In addition, there is a second special case observed in the *gemm* kernel, its code can be seen in Listing 5.2. Initially, several models were trained to check for differences in the training because of the randomness of both selecting the data for the training set and the shuffling of the data while training. Due to different signatures being used in the training and also different order when training, the obtained model can have different weights and biases, resulting in differences in the classification. In the majority of the created models, the *gemm* kernel was correctly classified as the pattern it has higher correlation with, the *Matrix x Vector* pattern. Nonetheless, in the case of a small number of models, this pattern was erroneously classified as a *Stencil* pattern. This problem appeared even with the current configuration of the ANN, which was obtained applying additional regularization techniques and an increase in the number of neurons to overcome overfitting problems and to diminish the lack of expressive ability with a limited number of trainable parameters.

The source of this problem was discovered looking at the correlation analysis between the patterns involved, where *gemm*'s kernel is highly correlated to both patterns instead of only to one of them. This relationship can be observed in the previous correlation analysis in Figure 5.7, where some other kernels are also highly correlated to more than one pattern.

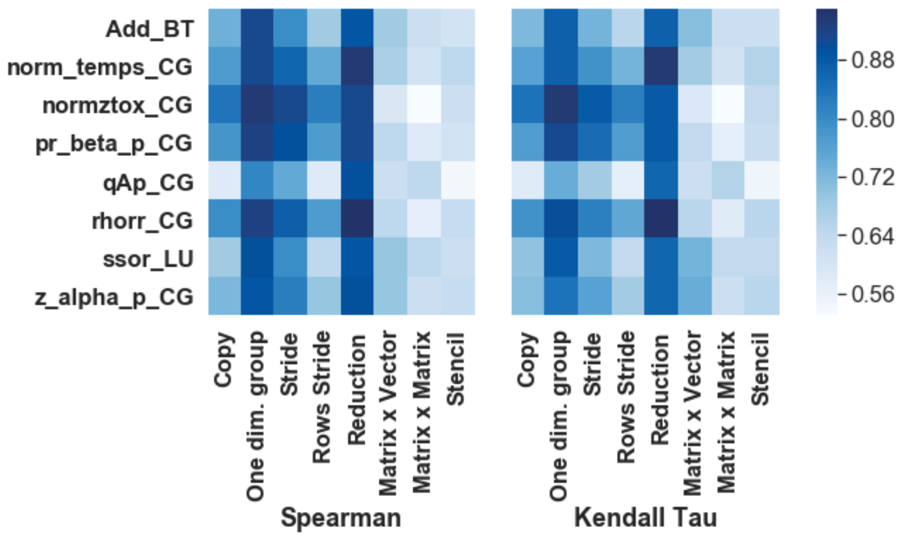


Figure 5.9: Correlation coefficients between NPB extracted kernels and the *pattern collection*.

```
#pragma omp parallel for private(i, j, k)
for (i = 0; i < _PB_NI; i++) {
    for (j = 0; j < _PB_NJ; j++)
        C[i][j]*=beta;
    for (k = 0; k < _PB_NK; k++)
        for (j = 0; j < _PB_NJ; j++)
          C[i][j]+=alpha*A[i][k]*B[k][j];
}
```

Listing 5.2: Code of the gemm kernel extracted from PolyBench.

Table 5.4: Classification of the kernels extracted from the NPB using the trained 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%)

The last part of the validation is performed with the kernels extracted from the NPB benchmarks. Table 5.4 shows the results produced by the pattern classification model for the 10 extracted kernels. In this case, we are proceeding the other way around because there is no previous correlation analysis that tells us to which pattern each of the kernels is highly correlated to. Consequently, to validate the classification done by the model, we have computed the correlation coefficients of each kernel signature to the patterns in the dataset. Figure 5.9 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 seen that the plotted results align with the classification given by the model. This results demonstrates that the ANN can also be used to detect candidate patterns no included yet in the dataset.

Finally, we have devised an experiment to show the importance of using a balanced training dataset. It consists of training an ANN model multiple times using an imbalanced dataset (discarded kernels are included in the pattern collection), validating the resulting model using some of the kernels extracted from NPB, and comparing the results to the previous results for the NPB benchmarks with the balanced dataset (see Table 5.4).

To simplify the experiment, the imbalanced dataset includes several dsicarded kernels of the *One Dimesional group*, *Stride* and *Stencil* patterns as independent classes and does not include kernels for the *Matrix x Vector* and *Matrix x Matrix* ones. We use the following kernels:

- Copy
- Add, Scale, Triad (form the One Dimensional group).
- Reduction.
- Stride2, Stride4, Stride 16 and Stride 64 (from Stride).
- 2PStencil and 2D4PStencil (from Stencil).

Table 5.5 shows the most frequent results produced by the ANN models trained using the unbalanced dataset. It can be seen that for the considered kernels that, in contrast with the previous results, the results using the imbalanced dataset are flipping between different possibilities, which for some cases belong to completely different patterns.

Table 5.5: Prediction given by the ANN model trained with the unbalanced set.

Kernel	**Prediction Unbalanced Set**
Add_BT	Add (65%), Scale (35%)
norm_temps_CG	Reduction (36%), 2D4PStencil (64%)
pr_beta_p_CG	2PStencil (31%), Triad (68%)
rhorr_CG	Reduction (46%), 2D4PStencil (54%)
rhs_norm_BT	Reduction (77%), 2D4PStencil (15%)
ssor_LU	Triad (32%), 2PStencil (62%)

5.8 Unsupervised learning results

In the previous subsection it was possible to see that a classification method based on supervised learning (an Artificial Neural Network) could successfully create models with hardware performance counters. Hence, a question arises, what happens when unsupervised learning methods are used?

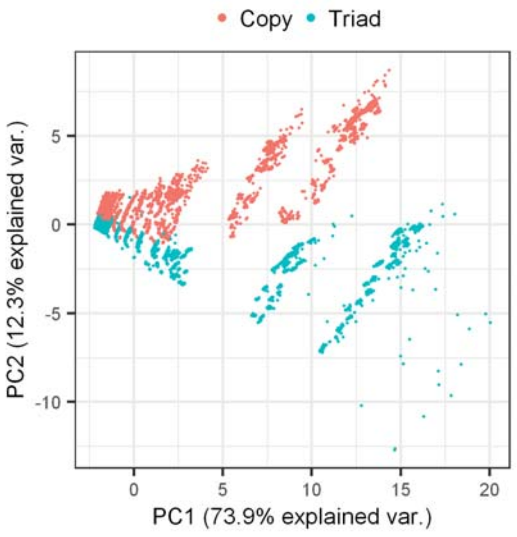


Figure 5.10: PCA for Copy and Triad from STREAM.

Figure 5.10 shows the PCA of Copy and Triad from the STREAM benchmark, which are included in the pattern collection. Each pattern is shown in a different color and point at the left of the PCA are small problem sizes, as we move to the right in the dimension defined by PC1, the problem size increases.

Looking at the PCA, it is easy to see that unsupervised method will likely perform classifications by problem size rather than by pattern.

One of the most well known unsupervised methods is K-means. This approach classifies data in different groups (called clusters) using centroids, which position is randomly generated and updated to minimize distances to the nearest elements to each centroid, at the end, each element is classified to the nearest centroid.

Figure 5.11 shows the result of applying K-means with two, three and four centroids. In the case of using only two centroids, the classification is performed in two clusters, one with small problem sizes and another with bigger problem sizes. The only difference of using a higher number of centroids is that more clusters of sizes are generated, as can be seen when using three and four clusters.

Consequently, the models generated with unsupervised learning were unable to classify by pattern. Instead, as unsupervised learning looks for similarity in the data, a classification can

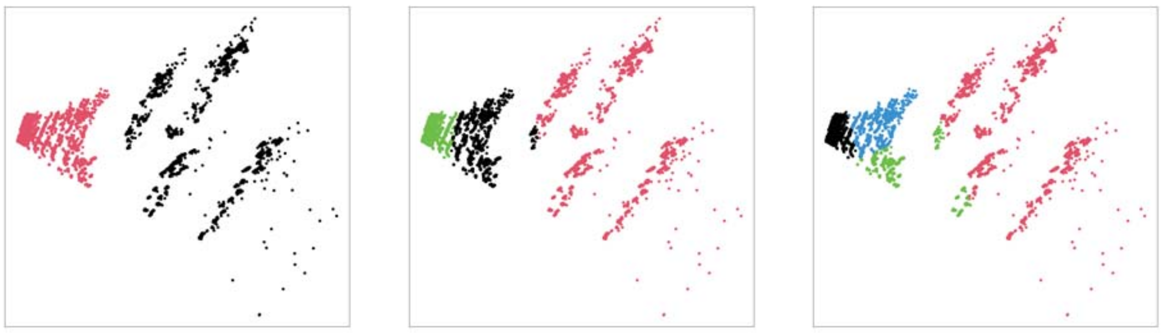

(a) Two clusters (b) Three clusters (c) Four clusters

Figure 5.11: PCA and correlation matrix for the reduced list of events.

be achieved per problem size, which may be useful to detect different workloads.

5.9 Conclusions

In this chapter a methodology to build datasets which can be used to generate models for performance tuning using machine learning has been developed.

The dataset is called **pattern collection** and should be both balanced and representative of patterns in OpenMP parallel regions. Each pattern in the collection should cover a unique portion of N-dimensional space represented by hardware performance counters.

The methodology assumes that the hardware performance counters used are non-redundant (the methodology from the former chapter has been applied to select them).

When a kernels wants to be integrated into the collection, the methodology is applied to check whether the candidate is a new pattern or it is already included in the collection. The methodology's step are:

- **Characterization of a candidate pattern**. The candidate is executed to obtain its signatures for two thread affinities (close and round robin) and from 1 thread (serial execution) to the maximum number of cores in the system.
- **Building candidate kernel representation**. A representation of the candidate where outliers are removed is generated. This representation only signatures normalized to the number of cycles and all the executions are included in a flattened vector.
- **Correlation analysis**. A correlation analysis is performed between the candidate and the pattern included in the dataset. If the candidate obtains correlation values higher than the predefined thresholds, the candidate is considered to be already covered by the patterns in the collection. Otherwise, the candidate is considered a new pattern to be included in the collection.

STREAM and PolyBench benchmarks were used to obtain kernels and provide candidates for the collection in two different systems. A total of 33 kernels were considered candidates and after applying the methodology, 8 patterns are found for Dell T7500 and 6 patterns in Dell PowerEdge R820.

The pattern collection was validated using an Artificial Neural Network which was trained using the patterns in the collection and the generated model was tested using the kernels discarded during the pattern collection's generation.

The validation is considered successful as the discarded candidates were classified correctly with an accuracy which ranges from 87% to 99%. Most of the cases incorrectly classified belong to the smallest problem sizes, where L1 and L2 are used, so there is less information because higher memories in the hierarchy are used. An additional reason is that OpenMP operations have higher impact in hardware performance counters in smaller problem sizes because execution times are short.

Furthermore, we tried to use unsupervised learning to classify the pattern collection. However, in the results we detected that the classification with unlabeled data generated a model which provided classifications closely related to problem sizes instead of OpenMP patterns.

The methodology to build balanced and representative datasets was published in [88].

Chapter 6

Dealing with Naturally Imbalanced Datasets

This chapter explains how some datasets are imbalanced by nature. In addition, some methods are presented to deal with imbalanced datasets when they are used to learn models in machine learning.

There are real problems where the obtained data is naturally imbalanced, such as fraud detection or medical data. In the case of fraud detection, a database is analyzed in [89], this dataset contains records with cases for automobile insurance with a data distribution of 6% fraudulent cases and 94% legit transactions. Other datasets for credit card fraud also show that fraudulent charges are a minority ([90] shows 2420 fraudulent cases in 31 million cases of the same operation type). This imbalance also appears in medical data, in [91] multiple datasets are described (see Table 6.1) which are naturally imbalanced as the incidence in each pathology is a minority, in the case of the WD dataset(breast cancer), less than 100 detected cases per each 100,000 patients.

Table 6.1: Characteristics of multiple medical datasets

Dataset Name	Total Instances	Minority Class	Majority Class
SP	267	55	212
MA	962	446	516
WD	569	212	357
CO	368	136	232
OST	313	85	228

In the same way, datasets for performance tuning can be naturally imbalanced due to characteristics of the hardware and/or the parallel programming paradigm, which may have a predilection towards a limited set of configuration values. Accordingly, because of this predilection towards some parameter configurations, the previous dataset may become imbalanced, because the dataset was designed to generate a balanced and representative collection of patterns. This imbalance can be clearly seen in Figure 6.4, where out of 12 thread configurations, one thread configuration is the ideal for approximately an 35% of the cases in the dataset.

As a consequence of the imbalance found in the dataset, machine learning techniques may underfit and not generate appropriate models because of incorrect inferences from the less represented configurations in the dataset [92]. Therefore, techniques to counter or reduce the effects of imbalance, when generating models using the dataset, are necessary for the automatic generation of performance models with machine learning.

There are three main types of methodologies that are applied to deal which imbalance datasets when using machine learning. These methodologies are classified in: **data**, **algorithmic** and **ensemble** [93]. All three methodologies are explained in the following sections.

Additionally, a study is presented to validate the accuracy of different models generated by these approaches for performance tuning against base models. The base models are the result of applying machine learning algorithms without balancing techniques.

6.1 Data methods

Data methods are based in re-sampling, the objective of re-sampling being to either increase or reduce the number of samples in a dataset.

The simplest methods are randomized under-sampling and over-sampling [94], that randomly select cases from the dataset. In the case of under-sampling, the chosen samples from the majority classes are removed from the dataset, while over-sampling duplicates the randomly chosen samples from the minority classes.

The main problem with random under-sampling is the loss of information because random cases are removed. Therefore, samples from which important inferences can be extracted are lost if unique (samples without duplicates) and representative cases are selected for removal, which can happen due to the random selection of samples.

In the case of random over-sampling, no new information is introduced, so the model tends to overfit. The overfitting is caused because of exact duplicates being used in the training, which diminishes the ability to generalize of the model.

To counteract the problems with random methods and achieve a balanced dataset, methods which generate synthetic data were developed. One of the initially proposed methods is called SMOTE [95] (Synthetic Minority Over-sampling Technique). This method over-samples the minority class using each minority class sample, generating synthethic samples along the lines joining the sample to each of its k closest neighbors from the same class. Its authors claim that this approach makes the minority class to become more general. SMOTE is a well-known over-sampling technique and other researchers have modified it to create multiple variants. For example, the python package called *smote-variants* includes 85 SMOTE variants [96], some of the variants change how samples are selected and/or how new samples are generated.

An example of SMOTE can be seen in Figure 6.1: (a) the initial dataset is shown where the minority class is represented with black dots and the majority class with red dots; in (b) the links between the different minority samples are shown. In this case, some parts of a few links are nearer to the majority class rather than the class they should represent, which makes it possible to create samples which could instead belong to the majority class; finally, in (c) synthetic data (gray dots) is randomly generated along the links and in some cases we have dubious data, which is clearly closest to the majority class, instead of the class they should belong to.

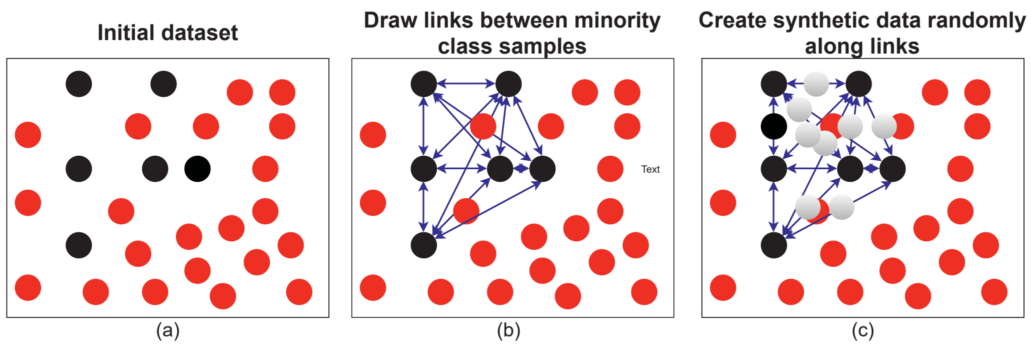

Initial dataset
Draw links between minority class samples
Create synthetic data randomly along links
Text
(a)
(b)
(c)

Figure 6.1: Example of SMOTE algorithm.

Another method to create synthetic data is Generative Adversarial Networks (GANs). This method is composed of two machine learning models called generator and discriminator [97] (an example can be seen in Figure 6.2). The discriminator is trained to detect whether the input data is real or synthetic, with an output of probability, and the result of the discriminator helps to improve the generator's synthetic data quality. The generator takes samples as input which is used to generate synthetic data, its objective is to generate samples with enough quality, in order to fool the discriminator and make it unable to identify it as synthetic. After the GANs is trained, the generator should be able to generate synthetic data which can help in training models with imbalanced datasets. This method is still in research and different architectures and modifications for this model can be found in different new works. These new works attempt to find an optimal configuration to provide valid data [98] [99]. The main challenge of this approach is finding the optimal architecture and parameters for both generator and discriminator, which may be different for different types of datasets.

6.2 Algorithmic methods

Algorithmic methods are modifications of already existing machine learning methods, that apply different approaches to counter the imbalance in the training data. Some of the approaches in this methodology are one class learning (or recognition based), threshold moving and cost sensitive

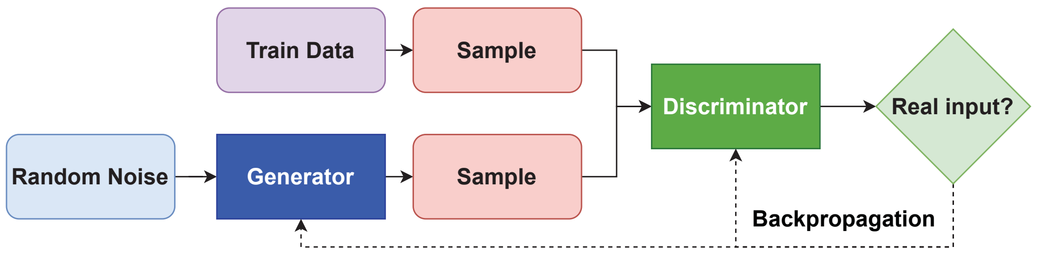


Figure 6.2: Example of Generative Adversarial Networks.

learning [93].

Cost sensitive learning is applied in some machine learning methods to change their error function. The new error function is based on the average cost of misclassification between the real class and the predicted class [100] [101]. The error function is represented as a cost matrix where each case has a different value and four cases are defined: true positive, false positive, false negative and true negative. Table 6.2 shows a simple example for a two class classification problem where predicting a sample correctly has no cost, whereas incorrect predictions have a predefined cost of 10.

Table 6.2: Simple cost matrix for binary problem

	Real Positive	**Real Negative**
Predicted Positive	*True positive*	*False positive*
Predicted Negative	*False negative*	*True Negative*

	Real Positive	**Real Negative**
Predicted Positive	0	10
Predicted Negative	10	0

A simple way to assign costs for each class is to use the method *compute_class_weight* from sklearn [102], which defines the matrix where weights are inversely proportional to the frequency of the data. The weight function can be seen in Eq. 6.1.

$$weight(class\_y) = \frac{total\_samples}{(n\_classes * samples(class\_y))} \tag{6.1}$$

In more complex cases, such as fraud detection, the cost matrix would be different as the costs of canceling credit cards, issuing new cards, reactivating cards and, most importantly, the fraudulent amount are involved.

6.3 Ensemble learning methods

Ensemble learning methods are the combination (ensemble) of multiple models, which can be generated by the same or by different machine learning algorithms. Ensemble models formulate a general hypothesis based on the combination of all the models' hypotheses [103].

A way to asses that each model formulates a different hypothesis, is to provide different data when training each individual model. A methodology called *Bootstrap Aggregation* is used to generate subsets of data (called bags) from a dataset to train ensemble models. This method randomly select different data points, which can be duplicated, from the original dataset. In this way, some models will be trained with different sets of data generating different hypotheses.

Another approach is an ensemble based on binary classification. Binary classification creates models where the prediction is between only two possibilities. Multi-class problems can be solved decomposing the problem into an ensemble of multiple binary classification problems with lower complexity [104]. An example of ensemble binary classification is one-against-rest, where each individual classifier predicts whether the input belongs to a class or not. In this way, there are as many classifiers as classes appear in the problem to predict, and each model is an expert detecting one class.

A well known ensemble method in machine learning is Random Forest. An ensemble of Decision Trees is called Random Forest as this method is composed of multiple individual Decision Trees. Each tree obtains a bag from *Bootstrap Aggregation* and for each tree at each node a random subset of features (metrics) are selected to perform the splitting of each node [105].

6.4 Analysis of machine learning techniques to automatically generate performance tuning models

Chapter 5 introduced a methodology to generate a balanced and representative dataset, which was used to generate a dataset of OpenMP parallel regions. In this dataset we added performance information about the best thread configuration for each combination of: problem size, pattern and thread affinity. The dataset with this additional information can be used to generate a model for tuning the number of threads in OpenMP parallel regions. However, there is a problem when the dataset is used for performance tuning.

The dataset was generated with the goal of generating a balanced and representative dataset of patterns. Therefore, when the dataset is used for a different purpose, such as performance tuning, this dataset may become imbalanced. This is because some parameter configurations, such as number of threads, are in general better than others. Consequently, if their frequency is checked to find which values are ideal, a few configurations are predominant while other configurations may not appear, or appear only in a low number of cases, compared to the most frequent configurations. This difference in the frequency explains why the dataset becomes imbalanced and generating a balanced dataset for such purpose can be impossible, as the ideal configuration cannot be known a priori, otherwise performance models would not be necessary at all.

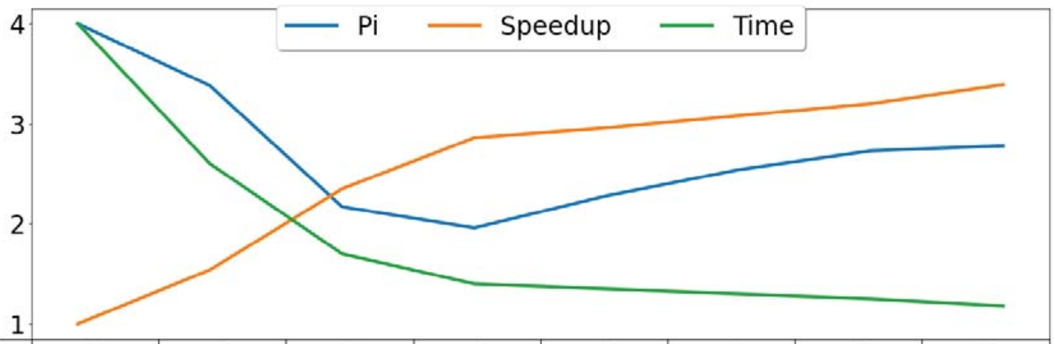
6.4.1 Predicting the ideal number of threads using basic ML techniques

We want to use the dataset to create a model using machine learning techniques with the objective of predicting the ideal number of threads. First, we need to define how the ideal number of threads is determined for a particular kernel in a certain configuration, each configuration described by both problem size and thread affinity.

The ideal number of threads is defined using a performance index described in [106]. This performance index is computed using Eq. 6.2, the performance index's value for a certain number of threads X is obtained dividing the execution time for X ($T_t(X)$) by its efficiency ($E(X)$). The ideal number of threads, using the performance index, is obtained with the number of threads (X) that minimizes the value ($P_i(X)$). The minimum value determines the optimum ratio between performance (execution time) in relation to resources (threads) the kernel is executed with.

$$Pi(X) = \frac{T_t(X)}{E(X)} = \frac{X \cdot T_t(X)^2}{T_t(1)} \tag{6.2}$$

Why this performance index is used instead of speedup to select the ideal number of threads is shown in Figure 6.3. In this example a computer with 8 threads is used and three metrics are shown: time, speedup and the the performance index's value. In this case it is clear that the maximum speedup is obtained with the maximum number of threads, however the decrease in time when using more than 4 threads does not compensate for the increase in resources. This is where the importance of using the performance index is highlighted, as the low decrease in time does not justify the use of more resources. Therefore, the performance index helps us determine the ideal number of threads to be used taking efficiency into account.



Threads	1	2	3	4	5	6	7	8
Time	4	2.6	1.7	1.4	1.35	1.3	1.25	1.18
Speedup	1	1.54	2.35	2.86	2.96	3.08	3.2	3.39
Pi	4	3.38	2.17	1.96	2.28	2.54	2.73	2.78

Figure 6.3: Relationship between execution time, speedup and the performance index

The performance index is applied for each execution configuration (pattern, problem size and thread affinity) in the dataset generated previously in the DELL T7500 and results are shown in Figure 6.4, where a clear imbalance in the ideal number of threads can be observed. If the dataset was balanced, each number of threads would appear with around 8.3% of occurrences. However, the most frequent thread configuration (12 threads) has a very high number of instances, with a percentage close to 36%, which is clearly several times bigger than the expected frequency for a balanced dataset. Additionally, the value with the lowest frequency is 7 threads, with a frequency of around 1.8%. Because of this clear imbalance, models will tend to have problems to learn the more under-represented configurations compared to the over-represented ones.

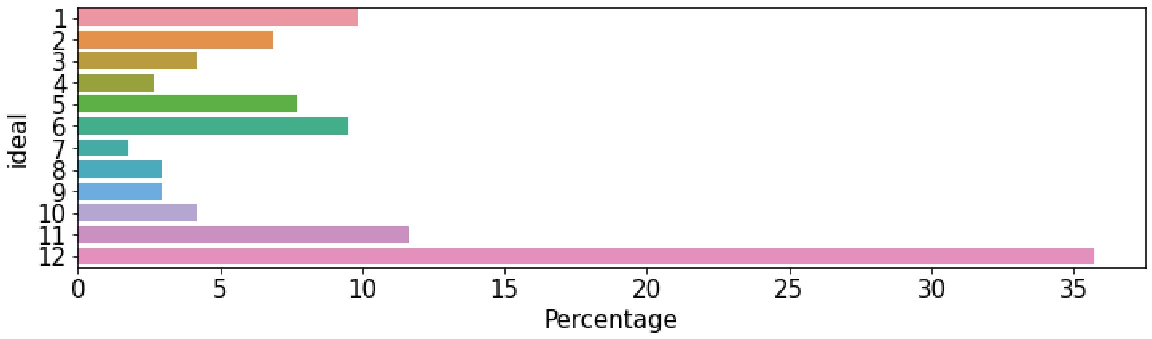


Figure 6.4: Ideal number of threads, according to $Pi(X)$, for all the executions stored in the dataset.

Because in this case the dataset is imbalanced and there is no way to generate a similar but balanced dataset, because the best value for a parameter that impacts performance cannot be known a priori, we consider that this dataset is naturally imbalanced. Therefore, the techniques previously explained in this chapter should be applied to generate appropriate models.

The first step when building models with ML is to find which machine learning algorithms are adequate for a dataset. In order to discriminate between different algorithms, we applied the following methodology with the pattern collection, which contains the ideal thread values, as the training set:

1. Select inputs. First, we need to decide which fields are selected as the inputs for the model to train. In this case the inputs are the hardware performance counters and the thread affinity the parallel region was executed with.

2. Select output. Once the inputs fields are selected, it is time to decide what the model should predict. In our case, to simplify the model, the ideal number of threads is the only parameter to predict.

3. Select machine learning algorithms to test. A ‘list of machine learning algorithms is selected, in this list the methods to call each of them are included. Additionally, parameters can be included to test some specific configurations. We have selected models such as Gaussian Naive Bayes, K nearest neighbours, Artificial Neural Network, Decision Trees and Random Forests and Logistic Regression.

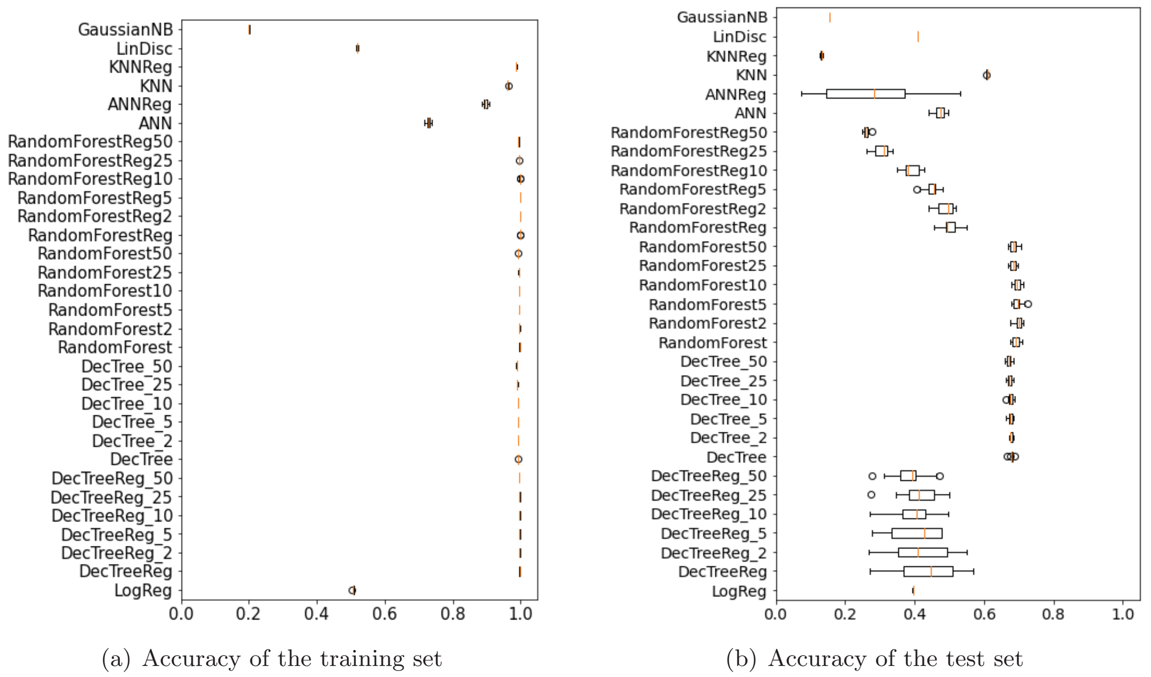
4. Apply Stratified K-Folds cross-validator to each model. The algorithms are executed mul-

tiple times to account for fluctuations in the training due random factors, such as data shuffling or the random split of the data for training and test, which can generate discrepancies when training models. The Stratified K-Folds cross-validator splits the dataset into K sets (folds), while trying to preserve the percentage of cases for each thread configuration in the output. We have selected five sets of K-Folds as this number allowed to find which algorithms show high variance in their models.

5. Plot and verify the results. Plot the results and select which algorithms generate better models for the dataset in use. The time spent when training the model may also be checked to compare. However, as once a model is trained this time does not matter, the training time is not as important as the accuracy of the model.

After applying the explained methodology, we obtained the results shown in the two box-plots of Figure 6.5. The training dataset is the pattern collection and the test dataset is composed of the pattern collection and, additionally, some of the discarded kernels that have been executed for all the problem sizes described in the previous chapter (Section 5.2).

Figure 6.5(a) shows the accuracy of the training set for each trained model. A first relevant observation in the training is that the accuracy in each algorithm does not suffer big fluctuations. The second significant result is that most models provide an accuracy close to 1 (which means 100% correct predictions), except for Gaussian Naive Bayes, Linear Discriminant Analysis (LinDisc) and Logistic Regression which are not even able to correctly predict 60% of the data used in the training.



(a) Accuracy of the training set

(b) Accuracy of the test set

Figure 6.5: Results after training different machine learning models.

The accuracy for the test dataset is shown in Figure 6.5(b). In this case, the best models in terms of accuracy reach almost 75% accuracy and with low fluctuations between the different trained folds. Making a simple calculation and taking into account that half of the kernels belong to the training set, we can obtain the accuracy for the unknown kernels $(2 \times test\_set\_accuracy) - train\_set\_accuracy = 2 \times 0.75 - 1 = 0.5$, which is an accuracy of 50% for the unknown kernels.

Although the result for Artificial Neural Networks may look underwhelming, a simple architecture was tested as scikit-learn uses by default only one hidden layer, so ANNs may provide better results with more complex architectures. Additionally, regression models provided results which are less accurate in the test and with high variability.

The good accuracy of K Nearest Neighbours (KNN) is surprising because although KNN is a simple algorithm, it is able to generate predictions with an accuracy close to that of Decision Trees with our dataset. However, one downside of this algorithm is the lack of parameters to tune for imbalanced datasets.

The best models for performance tuning are obtained with classification for Decision Trees, Random Forests (which is the ensemble of Decision Tree), Artificial Neural Networks and K nearest neighbours, according to the accuracy seen in Figure 6.5(b).

Now, we want to check how the models we found as the best for performance tuning behave if their parameters are modified. Balancing strategies will be applied in the next subsection.

The predictions for the models are shown in the following figures, where each sample may be classified in one of the following three categories:

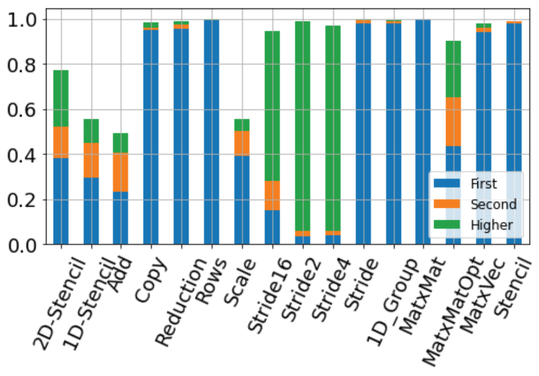
- First. In blue the ideal thread configuration is correctly predicted.
- Second. In orange the second ideal thread configuration using the performance index is predicted instead of the ideal.
- Higher. In green when, instead of predicting the first or second best configurations, the predicted configuration underestimates the number of threads. While predictions falling in this category are not the best, resource wise are good as, at least, resources are not wasted.

The machine learning algorithms to test, as they showed higher accuracy, are KNN, Decision Tress and Artificial Neural Networks.

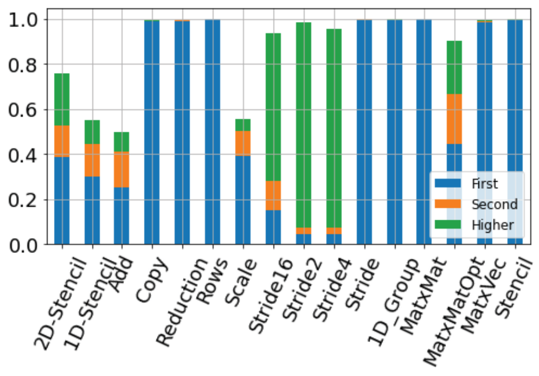
First, the simplest algorithm KNN is used and its results are verified for each kernel in the test set. The default configuration where K=5 is used as modifying this value did not provide more accurate predictions. The results are seen in Figure 6.6(a) where the default configuration with uniform weights for each *K=5* closest neighbours. In contrast, Figure 6.6(b) has weights which are inversely proportional to distance, therefore closer points are more important.

In both cases the accuracy for the training sets are close to 1 for the ideal thread. However, the kernels not included in the training have for the ideal configuration, in general, an accuracy which ranges from 0.4 to 0.03 in the smaller Strides. Stride predictions are clearly using less resources than the ideal configuration in a 90% of the predictions.

Results in this case are not outstanding but better than expected considering that KNN is a scheme similar to clustering and relationships between features are not analyzed.



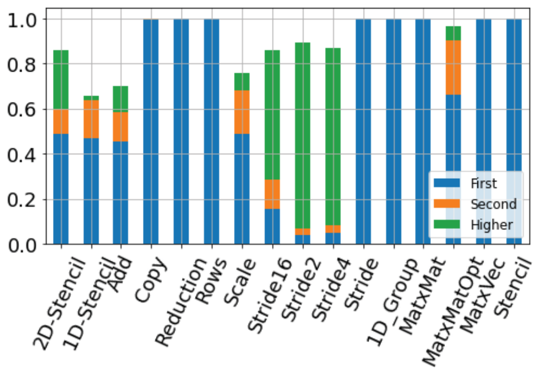
(a) KNN uniform weights



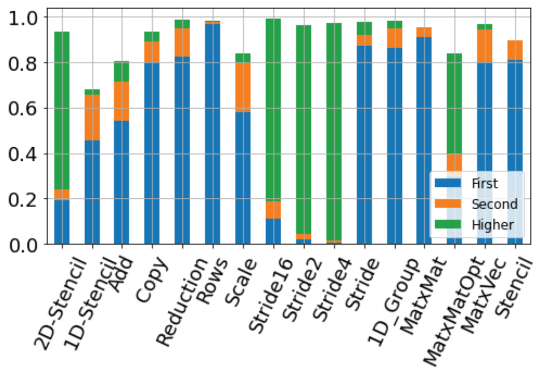
(b) KNN with weights inverse to distance

Figure 6.6: Results for models trained with K neares neighbours.

Now, Decision Trees and ANNs are used to generate models and compared to analyze how well they classify kernels with regards to the simpler KNN algorithm.



(a) Decision Tree



(b) Artificial Neural Network

Figure 6.7: Results for base models.

A model based on Decision Trees is shown in Figure 6.7(a). The parameters for the DT are the default parameters from *scikit-learn*, such as gini impurity for the splitting criterion, choose the best split between different splits and a minimum of two samples are required per split. The results obtained for this model provide more accurate results than KNNs, where 4 of the test kernels for the First and Second categories are correctly predicted for more than 60% of the samples. Additionally, the kernel called *Add* almost reaches 60% accuracy. However, there is one downside with Strides which, even with a better algorithm, remain a problem because the accuracy is extremely low for the First and Second categories.

Figure 6.7(b) shows the results obtained with a model trained with a multi-layer ANN. The architecture for which we found the best results is shown in Table 6.3 and was generated using AutoKeras [107], a framework to automatically generate and test artificial network architectures. The ANN has three hidden layer, where SELU is used as activation function with a dropout of 10%. In this case, although some test cases show better accuracy than Decision Trees, the obtained accuracy is generally worse for both the elements of the training set and the test set.

Table 6.3: Artificial Neural Network Architecture

Layer	**Neurons**	**Inputs**	**Activation**	**Dropout**
Input	N/A	19	N/A	0%
Hidden 1	19	19*19	SELU	10%
Hidden 2	200	19*200	SELU	10%
Hidden 3	200	200*200	SELU	10%
Output	12	200*12	Softmax	0%

6.4.2 Applying balancing techniques to basic ML techniques

The logical step now is to apply balancing techniques to the machine learning algorithms to counter the imbalance in the dataset. In this way, we assess the effect of using these techniques in the accuracy for both the training and the test sets.

First, data methods based on re-sampling are applied to our dataset and their results are analyzed. In re-sampling there are two categories: under-sampling and over-sampling.

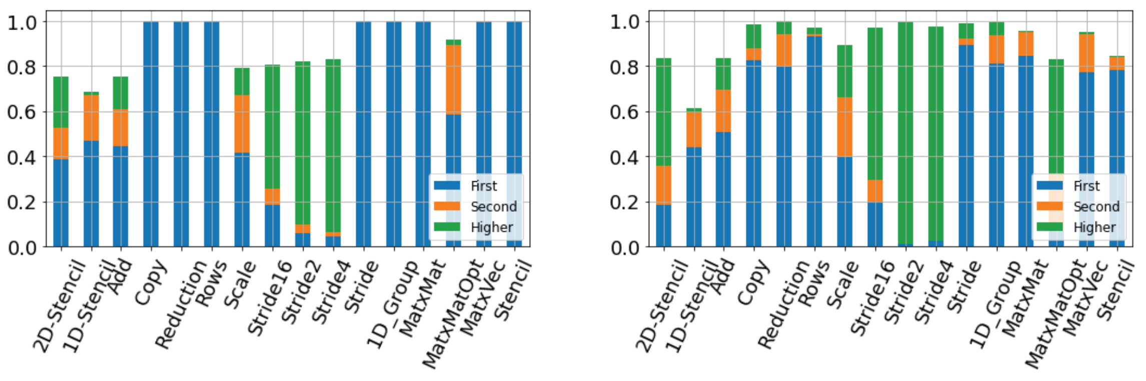
We abandoned the idea of using under-sampling as a lot of samples will be discarded according to the percentages seen in Figure 6.4, as the less represented case has around 1.6% sample, this will mean that only 1.6% of the data will be in each thread configuration after applying under-sampling. Therefore, the removal of samples results in a great loss of information as less than 20% of the samples remain.

Consequently, as under-sampling is discarded, when re-sampling the dataset there is only the possibility of applying over-sampling. With over-sampling, all labels will have the same number of samples as the most frequent thread configuration (12 threads). When applying over-sampling there are two main approaches: data replication and generation of synthetic data.

For data replication, a simple method is random over-sampling, however using this method in this case is a bad idea because there is a big difference between the highest percentage, so the less represented configurations will be replicated several times. This approach potentially entails that the model will not be able to generalize the relationships between features for the more under-represented configurations. Therefore, it is better to try to generate synthetic data instead of data replication.

In the case of synthetic data generation, there are two well-know methods: SMOTE and Generative Adversarial Networks.

Figure 6.8 shows the results for models trained after SMOTE has been applied to the dataset. In this case, models have been trained for both ANN and Decision Trees. Looking at the previous results shown in Figure 6.7, we were unable to find improvements after applying over-sampling to generate synthetic samples. In SMOTE samples are generated drawing lines between samples of the same class and selecting a random point in the line. Therefore, relations between metrics for a class are not taking into account, which potentially generates cases where incorrect inferences in the data are performed, because the synthetic samples may never naturally occur in reality. So SMOTE was discarded as further use of its synthetic data is detrimental.



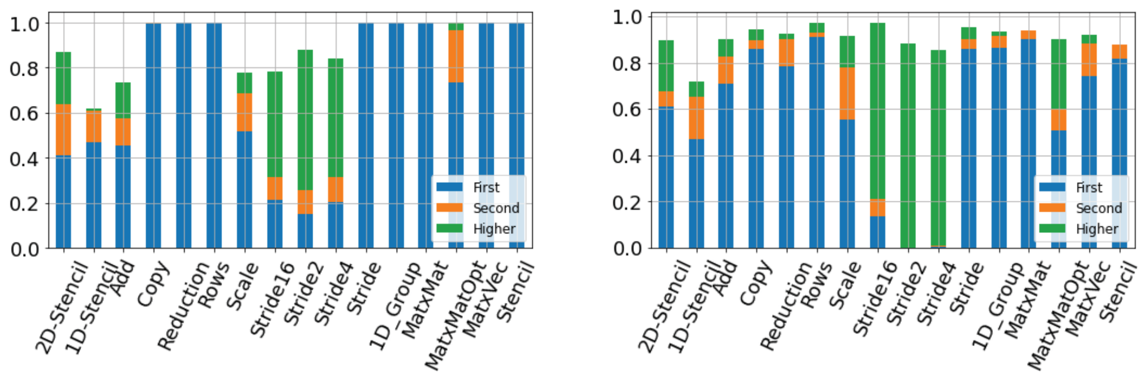
(a) Decision Tree (b) Artificial Neural Network

Figure 6.8: Predicting the number of threads balancing using SMOTE.

The second tested approach for over-sampling is Generative Adversarial Networks. An implementation from the python library called *CTGAN* [108], which is still under development, was initially tested. However, it provided inadequate results as some hardware performance counters appeared with negative values. Then, a second implementation called *CopulaGAN* was later found in the documentation, this implementation avoids the problem of negative values in hardware counters as data distributions can be applied when training the generator, therefore values for each field are generated inside the ranges provided by the data distributions. Because this approach can be considered more intelligent than SMOTE, we expect to obtain better results. One downside of this approach is the library being in the beta stage, so the use of big datasets, such as our case, should not be used because the training time for only one model in a 32 core machine needs more than a week.

The models trained with additional synthetic samples using *CopulaGAN* for both ANN and Decision Trees are shown in Figure 6.9 and provide results with higher accuracy than SMOTE, as can be seen in the Strides kernels. Both models outperform KNN cases, additionaly, when compared to the base models, the generated Decision Trees are similar with an accuracy's increase in Strides. In the case of ANNs, accuracy highly increases compared to the base ANN model. The problem with Strides still remain as the highest accuracy does not reach 30%.

This approach looks promising and should be further researched once *CTGAN* is not in beta or there are more libraries available to train GANs. Because the dataset has been modified and now it is balanced, additional balancing techniques cannot be applied.

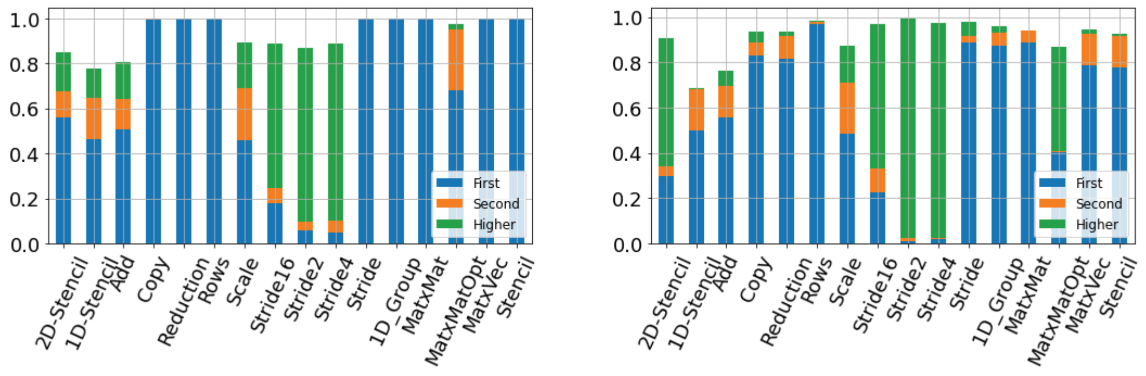


(a) Decision Tree (b) Artificial Neural Network

Figure 6.9: Predicting the number of threads using GANs.

After the data approach has been explored, the second step is to test how adequate cost-sensitive algorithms are to counter imbalance in our dataset. Both Artificial Neural Networks and Decision Trees have parameters to apply cost as weights, which are inversely proportional to configuration frequency in the dataset. In the case of ANNs it can be done in Keras [109] with the parameter *class_weight* to which a dictionary with the weights needs to be provided. On the other hand, Decision Trees automatically apply weights in *scikit learn* if the parameter $class\_weight = balanced$.

Figure 6.10 shows the results for both ANN and Decision Tree where only cost sensitive learning has been applied to the configuration of the base models of Figure 6.7. Both models show results which are better than the base model. More improvement can be seen in the ANN model, however Decision Trees in our case seem to generate more accurate models without synthetic data.



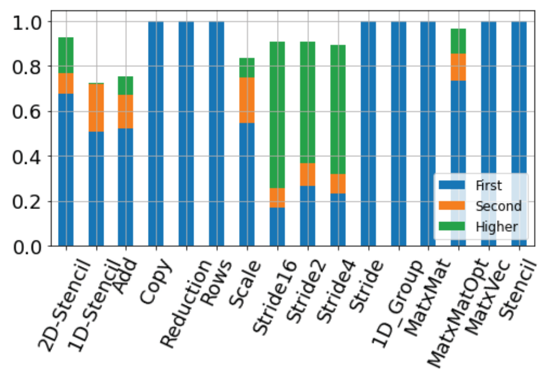
(a) Decision Tree (b) Artificial Neural Network

Figure 6.10: Predicting the number of threads with cost-sensitive learning.

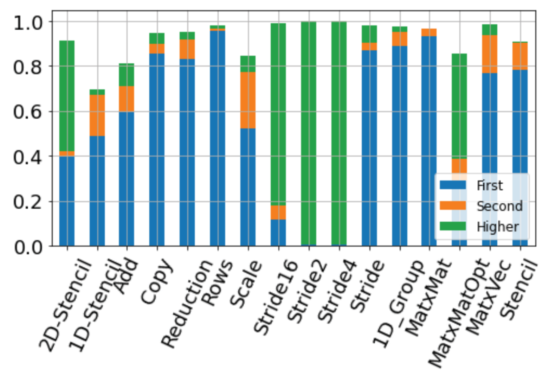
Results look promising because the models' accuracy improve when balancing techniques are applied. Therefore, the models may further improve if the adequate approaches are explored.

The last step is to apply ensemble learning methods, in the case of Decision Trees their evolution into ensemble are Random Forests, which are a combination of different trees where each tree explores a set of metrics. In the case of ANNs, the *BaggingClassifier* from scikit learn is applied and one network is trained for each input metric, for a total of 19 networks (18 hardware performance counters and one additional for thread affinity). *Bootstrap Aggregation* is applied and each individual tree and network makes use of a different subset of the training dataset, called bag. This introduces some randomness in the generation of the models as the subsets are generated selecting data randomly. Therefore, each time a model is trained it may provide different results.

In Figure 6.11, where ensemble learning is applied, results for the ANN (Figure 6.11(b)) do not improve and are significantly worse than the ensemble Random Forest, although the training time needed is around 20 times higher than training a single ANN (approximately 10 hours for the ensemble). In the case of Random Forest (Figure 6.11(a)) the accuracy is improved, the best improvement can be seen in the Strides, which are the most problematic kernels, and in some cases the accuracy triplicates compared to the cost-sensitive approach with Decision Trees. The configuration parameters for the Random Forest are the same for the common parameters with the previous Decision Trees, the additional parameter is the number of estimators, which is set to a hundred trees.



(a) Random Forest

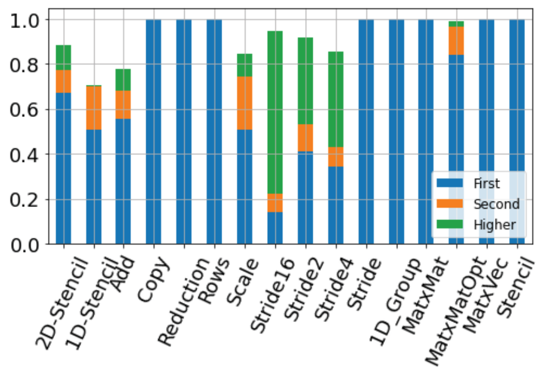


(b) Artificial Neural Network

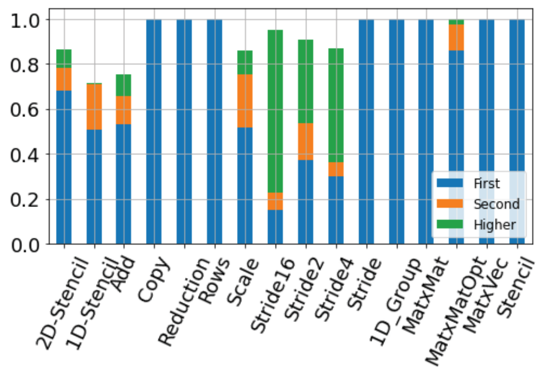
Figure 6.11: Predicting the number of threads with ensemble models.

The final tests were made using Random Forest with binary classification, with and without applying cost-sensitive learning. With binary classification the problem is divided into 12 different sub-problems where each trained model detects whether samples belong to one particular thread configuration or not. The models for this test with ANNs are not considered as the accuracy remains lower than Decision Trees.

Binary classification has improved accuracy, as seen in Figure 6.12(a), compared to the previous Random Forest with default values. The best achievement with binary classification is that two Stride cases have accuracy higher than 40% using the best two possible configurations for the performance index. Additionally, a test case where matrix multiplications are represented reaches an accuracy of 96%, while other cases, without taking into account Strides, have an accuracy higher than 70%. Figure 6.12 shows that applying cost sensitive learning to binary classification does not improve the obtained results.



(a) Binary Random Forest



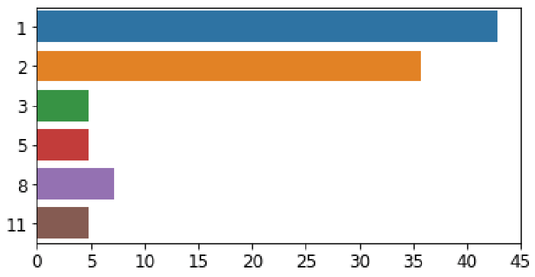
(b) Binary Random Forest with cost

Figure 6.12: Predicting the number of threads with Random Forest binary classification.

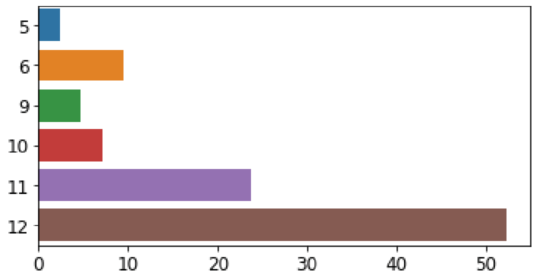
If the results for Stride are not taken into account, the accuracy of the generated models seem appropriate, but Strides remain to be a thorn in our side since the beginning. The explanation for these kernels is found after the Stride kernels and the Stride pattern are analyzed.

If we take a look at Figure 6.4 with the ideal thread configurations, the most over-represented thread configuration is 12, so the case of the different Strides, which in most cases show that the prediction select a lower number of threads than the ideal, can be counter-intuitive. Because this issue appears in all tested models, be it KNN, Decision Trees or Neural Networks, and also even when balancing techniques are used, a thing to look at is the pattern they are closer to in the correlation analysis. The problematic cases are similar to the Stride pattern which is a loop region with jumps (strides) in memory accesses of 64 positions, while the problematic kernels have strides in the order of: 2, 4 and 16. If we take a look at the thread configurations of the Stride pattern (Figure 6.13(a)), it is clear that this pattern must be executed with only one or two threads, so parallelism in this pattern is not a good approach. Consequently, the question here is: **Does the same apply to the problematic Stride kernels?**

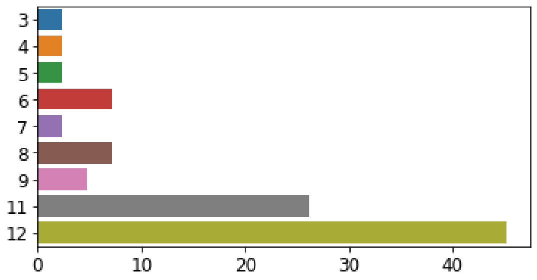
In the case of the kernels the same does not apply, because the serial or two thread configuration does only appear as an ideal configuration for Stride16 in a low percentage of the cases (less than 10%). In the cases of Stride2 and Stride4 kernels, the best approaches are to use 11 and 12 threads. So the main problem here is that lower strides make better use of cache than the pattern, therefore the ideal thread configurations can be used efficiently with more resources. However, the pattern they are more similar to is not as efficient, consequently the predictions of the models tend to choose thread configuration with a lower number of threads. An experiment to confirm this hypothesis is to include Stride4 in training dataset and then train a new model to see if the predictions for Stride2, which shows a similar thread distribution, improve.



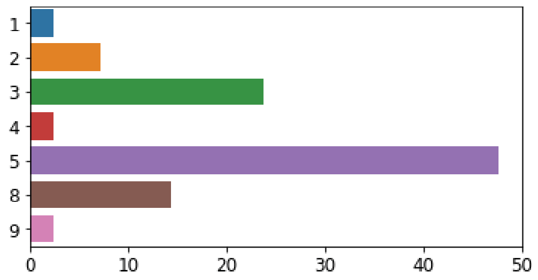
(a) Stride pattern thread configurations



(b) Stride2 thread configurations



(c) Stride4 thread configurations



(d) Stride16 thread configurations

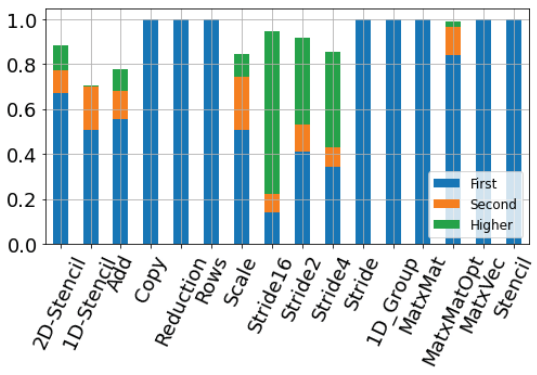
Figure 6.13: Percentage of samples in each thread configuration for Stride pattern and kernels.

In Figure 6.14 the hypothesis is tested comparing the former binary classification with Random Forest and comparing it to the same configuration but now the Stride4 is also in the training set. Looking at Figure 6.14(b) it is clear that the accuracy for Stride2 has increased for more than 20%, and now it reaches an accuracy of 65% for the ideal thread configuration. Additionally, in the case of including the second possible configuration, accuracy is around 83.4% correct predictions. However, the accuracy for Stride16 has not been affected as the ideal thread configurations for this kernel are not similar to Stride4 configurations, 3 or 5 threads is in general the best configurations for Stride16 whereas 11 or 12 threads are ideal for Stride4.

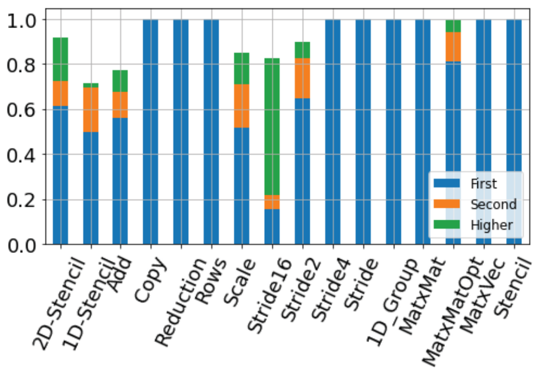
Additionally, the importance of each input feature used in the training dataset has been researched. There are two ways to obtain the importance of a feature for Decision Trees [110]:

- Impurity-based importance, also called as mean decrease in impurity, should be avoided as it suffers of biases when numerical features with high cardinality are used. Additionally, this approach cannot be used in data not used in the training.

- Permutation feature importance. Permutations are performed in the features and miss-classifications when a metric is permuted are used to compute importance. Permutations in the most important metrics induce to worse classifications in the model.



(a) Binary Random Forest

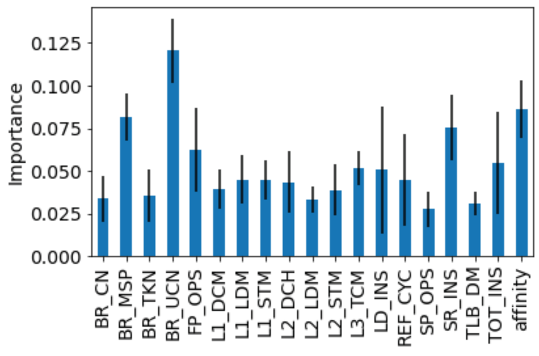


(b) Binary Random Forest with Stride4

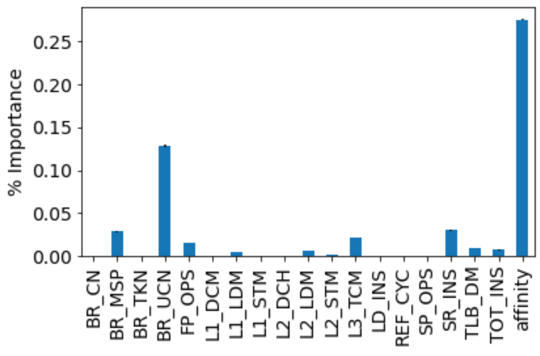
Figure 6.14: Predicting the number of threads to confirm Stride hypothesis.

Both approaches are employed to calculate the importance of each feature and their importance can be seen in Figure 6.15 for a Random Forest model. First, Figure 6.15(a) shows the impurity-based, but it should not be trusted because of the biases when numerical features with high cardinality are used, so this case is only used to check how similar it is to the permutation approach. Figure 6.15(b) shows the importance for each feature when using permutations.

The most important features according to the importance analysis with permutations are affinity and unconditional branches, the importance of affinity defining the number of threads cannot be denied. However, the importance given to each hardware performance counter is unexpected as cache, load and stores show low importance in the model to decide the ideal number of threads.



(a) Impurity importance



(b) Permutation importance

Figure 6.15: Features importance for Random Forest.

As shown in the experimentation, balancing techniques are very important when creating models to predict the ideal tuning parameters values because balanced and representative datasets may become imbalanced for the target parameter to tune.

The most encouraging results were found with two approaches:

- Synthetic data. The use of CopulaGAN has provided promising results. This approach should be researched in the future if more implementations are included in machine learning libraries.
- Ensemble binary classification with Random Forest. The use of binary classification has greatly increased accuracy in the most troublesome cases (Strides) as the generated models are more specialized, because each model is trained to only detect a particular thread configuration.

Results in Dell PowerEdge R820

A summary of the results obtained in Dell PowerEdge R820 are described hereunder.

First of all, in this system there are 32 cores, as opposed to the Dell T7500 with 12 cores. In the same way, 29 problem sizes are necessary instead of 21 due to the increase of cores and processors in the system. Therefore, the number of samples necessary to obtain all the signatures composing each pattern is approximately 4 times bigger, generating a bigger dataset.

The thread distribution using the performance index is calculated and shown in Figure 6.16. Although the difference in the percentage in the two thread configurations with more samples is not as high as in the previous system, there is a clear imbalance in the dataset. The case with more ocurrences (32 threads) contains around 18% of all the samples, which is 56 times higher than the case with 21 threads with only 0.32% of the samples, a bigger imbalance than the previous system.

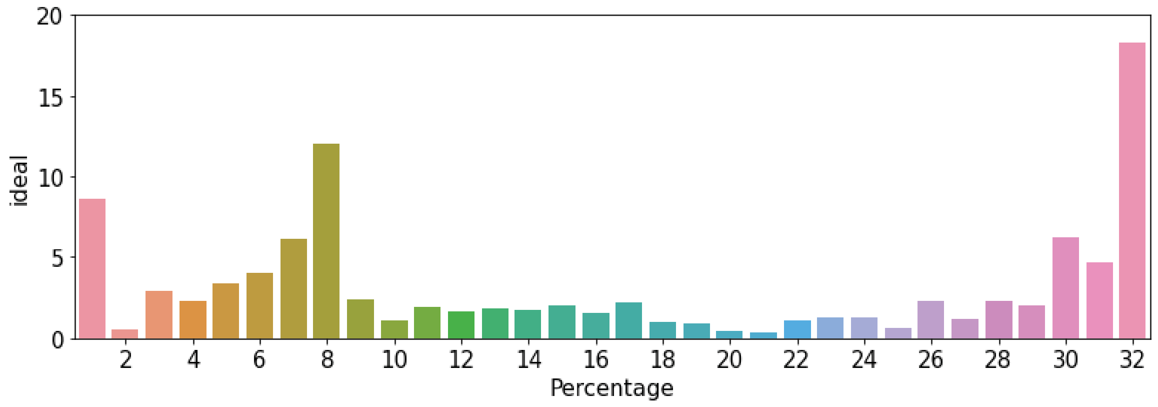
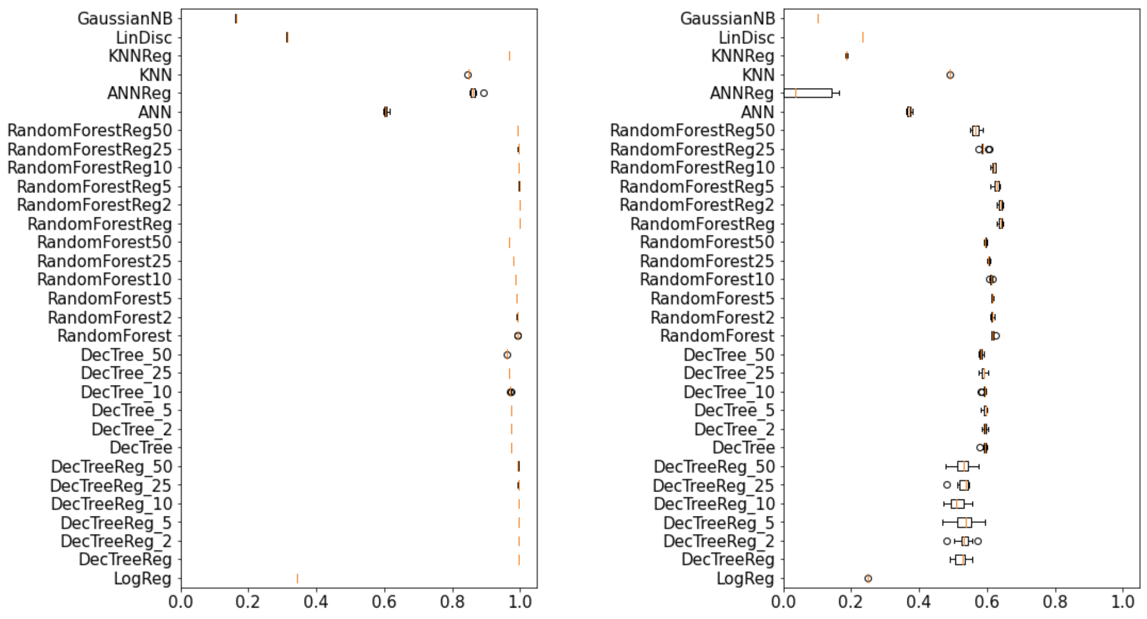


Figure 6.16: Ideal number of threads according to $Pi(X)$ for R820.

For this system the same methodology using the K-Folds cross-validator to discriminate between different machine learning algorithms is applied. The training set is composed of the pattern collection and the test includes both the pattern collection and some of the discarded kernels for all the problem sizes in the collection.

We obtained the results seen in the two box-plots of Figure 6.17, where we see that using regression based models for Random Forests and Decision Trees provide similar results to clas-



(a) Accuracy training (b) Accuracy all dataset

Figure 6.17: Results after training different machine learning models in R820.

sification models. Additionally, in some cases of Random Forests their accuracy is higher, but have more variability in the results. Therefore, as balancing techniques cannot be applied to regression based models and the difference between classification and regression for this system is negligible, the study is performed for classification models.

We can also see that KNN performs a bit worse than tree based approaches. We wanted to check the results by kernel for this model but it not feasible. The predictions for all the KNN classification models took 150 hours, this is 6 days and 6 hours. Making predictions with the model with all samples will take similar times. As better models with better prediction time are available, we discarded this model.

Figure 6.18 shows the time necessary to predict a sample. It can be seen that the time to predict a sample is negligible. In the case of KNN, it is important to take into account that prediction time is directly proportional to the number of samples the model is generated with, which makes it a bad approach when big datasets are used.

Now, Decision Trees and Artificial Neural Networks are used to generate models without applying balanced techniques and obtain two cases which are used as base models. Figure 6.19 shows the base models built using the dataset for R820. In this case there are 6 patterns in the collection and in the case of the Decision Tree their accuracy is 100%. However, the same cannot be said for the Artificial Neural Network, where the accuracy for the training cases ranges from 60% to 97%. Furthermore, in the test cases, the accuracy of both models for two best thread configurations does not reach 50% in most cases.

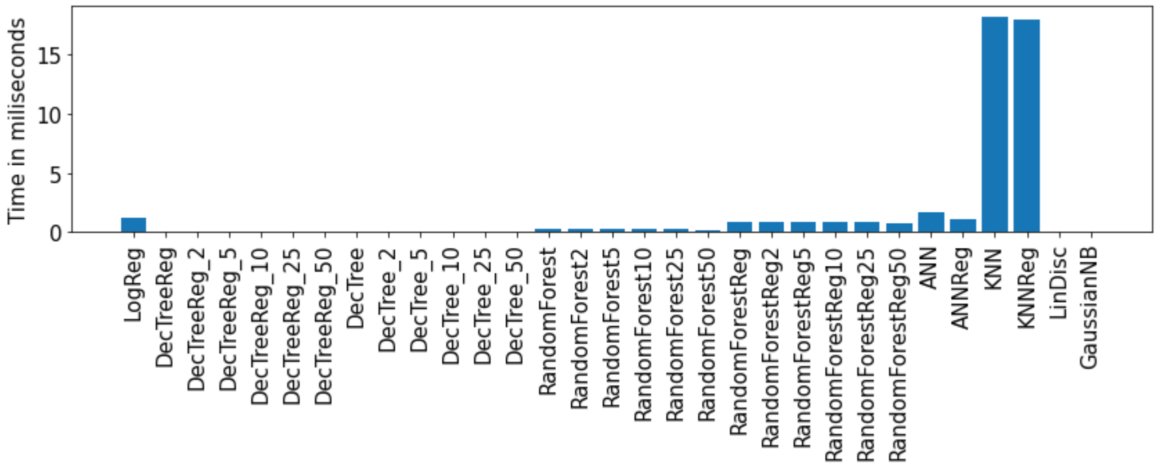
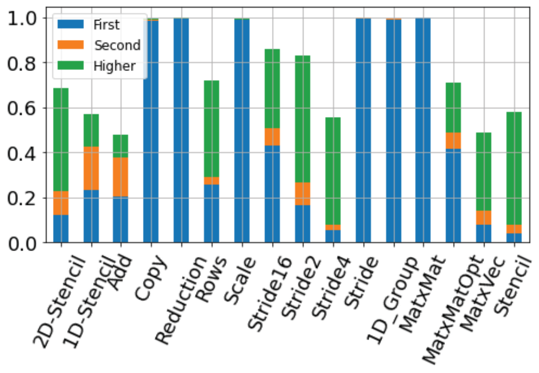
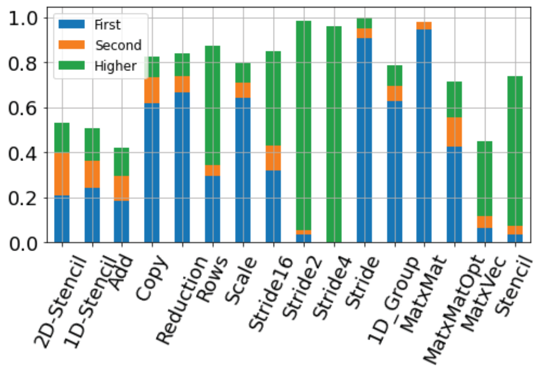


Figure 6.18: Prediction time per sample.



(a) Decision Tree

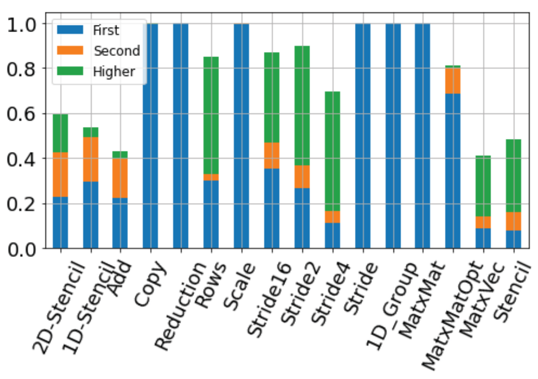


(b) Artificial Neural Network

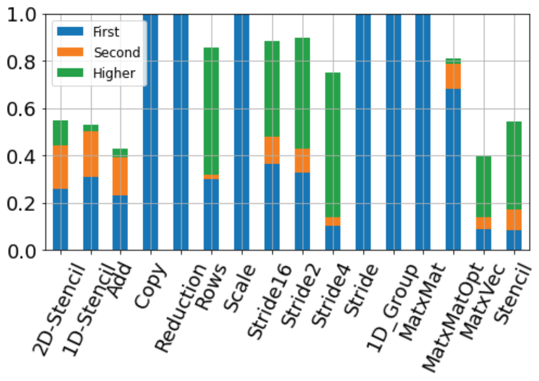
Figure 6.19: Results for base models without balancing in R820.

In the case of balancing techniques, data techniques were discarded as SMOTE provided even worse results. Additionally, we were not able to train GANs because the libraries are still under development and generated errors in memory due to the size of the dataset.

Then, the next step is to test cost-sensitive algorithms and ensemble learning methods. Better models were generated using an ensemble of binary Random Forest with and without cost-sensitive learning. Figure 6.20 shows the results for the best models generated, which show, in general, bad results compared to the previous system. Additional steps or alternative approaches are necessary in this case as there number of classes to predict almost triplicate and the imbalance in the dataset is also greater, which causes the model to fail in providing accurate predictions.



(a) Binary Random Forest



(b) Binary Random Forest with cost

Figure 6.20: Results for the best generated models in R820.

6.5 Conclusions

In this chapter a study on how to build machine learning models with imbalanced data was performed. This study is very important because due to system characteristics some parameter configurations are, in general, better than other configurations. Therefore, datasets for performance tuning are naturally imbalanced as some configurations will naturally have more samples than others. This imbalance in the data can have a negative impact on the generated model as the most represented classes (each configuration is considered a class in ML) have more information to infer relationships from, so the underrepresented classes have a clear weakness.

Techniques to counter the imbalance the less represented cases are under should be used. Some techniques are explained in this chapter, divided in:

- Data methods. The dataset is modified with the use of techniques to reduce the samples in the more represented class, replicate data in the less represented classes or the use of algorithms to generate synthetic data.
- Algorithmic methods. Modifications to machine learning methods are applied to counter imbalance, such as applying weights inversely proportional to the frequency of each class under consideration.
- Ensemble learning methods. Multiple machine learning models are combined into a single model to generate more hypotheses.

Different supervised machine learning techniques were trained to find the ideal number of threads for multiple kernels executed for multiple different problem sizes. The ideal number of thread is calculated using the performance index described in [106] which finds an optimum between performance and resources.

In initial results without applying balancing techniques, the accuracy of the models was perfect or almost perfect for Decision Tress and Artificial Neural Networks. But in the case of the test data, the percentage of correct predictions for the ideal number of threads was, in general, bellow 50%.

Then, data methods to generate synthetic data were applied, such as SMOTE and Generative Adversarial Networks. Results improved from the initial models with the use of GANs while SMOTE did not provide any improvement. GANs looks like a promising methodology to improve accuracy but architectures for this approach are still in research, so data generated with future architectures may further help to increase accuracy.

Afterwards, algorithm modifications and ensemble techniques were applied to the dataset without synthetic data. Newer models with these techniques generated models with higher accuracy. One of the best cases, ensemble binary classification with Random Forest, provided a general accuracy higher than 70% when stride kernels were not taken into account.

Stride kernels are treated as problematic kernels and an analysis to find why their accuracy was lower than other kernels was performed. The cause of this problem was that the kernel they are more similar to did not follow the same ideal thread distribution, consequently models underestimate their ideal thread configuration. A solution to this problem is including additional strides in the training set with similar thread distributions.

Results in Dell PowerEdge R820 are underwhelming because of more possible thread configurations, which generate bigger differences in the number of samples for each configuration. As a consequence, great imbalances in the dataset are found when used for performance tuning. Solving this problem is one of the open future lines of work as it could be solved with newer machine learning models or with modifications in the way the dataset is generated.

Chapter 7

Conclusions and Future Work

This chapter is divided in two parts, first the conclusions and main results of the work presented in this thesis are summarized, and, secondly, the future work where some open lines of this work are discussed.

7.1 Conclusions

This work has researched the possibility of using machine learning to generate models that can be applied to automatically tune the performance of parallel applications. The machine learning approach is used in other works, but, to the best of our knowledge, this is the first time a systematical set of methodologies are defined to determine the appropriate set of features for characterizing shared memory parallel regions and generating representative and balanced datasets.

When using machine learning to generate models the principle called "*garbage in, garbage out*" is fundamental. According to this principle, the model will only be as good as the data used in its training, so the proper data (a valid dataset obtained from different scenarios with the correct inputs) is necessary to build useful models. In the case of performance tuning the following two questions need to be answered: 1) there is a big number of measurable metrics (inputs) in parallel applications, how can the execution of an application be properly represented with a limited number of metrics?; 2) representative parallel regions executed in different scenarios should be used to generate a dataset, how can a representative and balanced dataset be built? Two methodologies, each answering one question, are defined in this thesis.

Out initial hypothesis was that the execution of shared memory parallel regions could be characterized by the values of a set of hardware performance counters. Consequently, as an initial step before building these methodologies, it was necessary to be able to collect the value of hardware performance counters for the parallel regions of an application. This was achieved by integrating PAPI[2] into MATE[44]. The integration was validated using a performance model described in [11], and the integration was published in: **Dynamic Tuning of OpenMP Memory Bound Applications in Multisocket Systems using MATE. ICPP Workshops 2018, 37:1-37:10. Jordi Alcaraz, Anna Sikora, Eduardo Cesar**

The first methodology defined in this thesis (Chapter 4) allows for determining the minimum set of hardware performance counters necessary to generate signatures of OpenMP parallel regions. This is achieved by performing a redundancy analysis based on correlation and PCA of the performance counters' values collected for a set of parallel code regions. This analysis can be extended to include additional metrics if their values are numerical or they can be converted into numerical values.

The summary of this methodology is that a correlation analysis is performed between the performance counters and in the case the correlation value between two metrics is above a threshold, the metric is considered as a candidate to be discarded. Because high correlation in hardware performance counters can appear due to code characteristics instead of redundancy in the metrics, logic should also be used to avoid discarding metrics which in reality are non-redundant. With this methodology the list of all-purpose events extracted from PAPI for the nodes used in this thesis has been reduced by more than half. An Artificial Neural Network was trained to check if the reduced list of events is able to characterize OpenMP parallel regions and the trained model obtained an accuracy close to 100%, which confirms that the reduction was successful. Published in: **Hardware Counters' Space Reduction for Code Region Characterization. Euro-Par 2019, pages 74-86. Jordi Alcaraz, Anna Sikora, Eduardo Cesar**

A second methodology was proposed in this thesis (Chapter 5) for generating balanced and representative pattern collections that can be used as a dataset to train performance models. Similarly to the previous methodology, correlation analysis is also employed. In this case two different correlation methods are applied to make the process more robust.

Given a pattern collection and a set of candidate kernels, a problem size minimizing the parallelization overhead is selected for each kernel and each candidate is executed for different thread configurations with close and spread affinity to obtain their signatures. Once the signatures of the candidates are collected, the correlation analysis is performed and candidates which have a correlation value higher than the threshold for both methods, are discarded as they are considered similar to a pattern already included in the collection. On the other hand, if both threshold are not surpassed, the candidate is considered to cover a new part of the search space and should to be included in the collection. Consequently, it must be executed for different problem sizes which characterize the multiple memory levels of the node. The rationale to select the problem sizes to be used is also explained in the same chapter. This methodology was used in two different nodes with a set of kernels extracted from STREAM and Polybench to create an example pattern collection. The kernels not included in the collection and a new set of kernels extracted from NPB were successfully classified by an Artificial Neural

Network trained using the example generated collection. Published in: **Building representative and balanced datasets of OpenMP parallel regions. PDP 2021, pages 67-74. Jordi Alcaraz, Steven Sleder, Ali Tehrani Jamsaz, Anna Sikora, Ali Jannesari, Joan Sorribes, Eduardo César**

Finally, a study to analyze how a pattern collection generated using the previous methodology can be used to automatically generate models for performance tuning is described in Chapter 6. One problem that may appear even for a balanced and representative pattern collection is that it can be naturally imbalanced for performance tuning due to characteristics of the hardware and/or the parallel programming paradigm, which may have a predilection towards a limited set of configuration values. In this study, base models are built using K nearest neighbours, Decision Trees and Artificial Neural Networks because they provided better accuracy than other machine learning algorithms analyzed. However, the accuracy obtained with these methods could be improved if the appropriate techniques for tackling the imbalance are applied.

Thus, balancing techniques are then applied to the machine learning approaches in order to see if the imbalance problem can be overcome and accuracy can be increased. In the case of data methods, good results were only obtained with Generative Adversarial Networks, which provided a small increase in accuracy but it looks promising as a proper architecture was not tested because this method is still in research. Then, algorithmic methods (cost-sensitive learning) were used to generate models with a general accuracy around 60%. However, there are three problematic kernels were one has an accuracy around 20% and the two with only 10%. Finally, ensemble methods were implemented, the use of multiple Random Forest (an ensemble of Decision Trees) as a binary classifier (ensemble of N Random Forests, N being the number of threads) provided the best results with a general accuracy higher than 70%. Furthermore, the three problematic cases also improved, two cases with accuracy higher than 40% and another around 20%. Paper in second revision: **Predicting Number of Threads using Balanced Datasets for OpenMP regions. PDP Special Issue 2021 in Computing, Springer. Jordi Alcaraz, Ali Tehrani Jamsaz, Akash Dutta, Anna Sikora, Ali Jannesari, Joan Sorribes, Eduardo César**

The results obtained in this thesis demonstrate that the proposed methodologies are a promising way for generating representative and balanced datasets which can be used for tuning the performance of parallel applications.

7.2 Future Work

There are several open lines which can be explored in the future to increase the robustness of the methodologies proposed in this thesis. In addition, a thorough exploration of the parameter of some of the machine learning algorithms used in this work would be necessary for improving the accuracy of the generated models.

The main future lines of work are:

- **Include additional metrics**. In our work only hardware performance metrics and thread affinity has been used as metrics for our models. Additional metrics, such as code characteristics or data dependencies could also be used to provide more information about the performance of a parallel region, as additional data is included explaining the behaviour of either another part of the system or characteristics of parallel region.
- **Improve the pattern collection generation methodology**. When applying correlation analysis between the candidate and the pattern collection, the candidate is not evaluated to find whether the kernel can be considered compute or memory bound. In this way the correlation analysis could also take into account which metrics are related to computation and which to memory to give them more or less weights depending on the behaviour of the kernel.
- **Implement better models**. Nowadays, the selection of the parameters when training machine learning models is not clearly defined and most approaches to find more accurate models consist in modifying the configuration parameters and test them instead of applying a methodology to select the correct parameters. If guidelines appear in machine learning to select the appropriate configurations parameters, proper model configurations could be found more easily and less time would be spent trying possible configurations.
- **Explore Generative Adversarial Networks**. Neural networks to generate synthetic data are still under research as they are state of the art approaches, so the adequate architecture for both the generator and the discriminator, the number of training steps and additional parameters to create good synthetic data are not known. In the future, either a method to select good parameter configurations may be developed and results may improve as the parameters are tailored to the characteristics of the dataset or a general combination of parameters for GANs may be found to generate synthetic data for different datasets.
- **Remove architecture's dependency**. The proposed approach is architecture dependent, therefore a model generated for a system will not work in another system with different hardware characteristics. There are some works, such as [111] [112], explaining initial approaches to generate portable performance metrics, which could be used to remove the architecture dependency when generating performance models with machine learning.
- **Extend methodologies to other paradigms**. The proposed methodologies were developed for OpenMP parallel applications, therefore to use the methodologies for other parallel paradigms, such as multi-node (MPI) systems or accelerators (CUDA, OpenCL), modifications may be necessary to properly generate balanced and representative datasets for performance tuning.

- **Apply tuning to additional parameters**. In the study only the number of threads was predicted, however there are additional tunable parameters in OpenMP such as the scheduling type and its chunk size. Additional models could be generated to cover the different tuning parameters or multi-output learning [113] could be implemented to tackle multiple parameters with a single model.
- **Implement models into dynamic performance tools**. The machine learning models generated in this thesis could be implemented into tools to apply dynamic performance tuning. First, as most performance tools are implemented using either C or C++, the models should be exported from Python and the proper libraries should be found to correctly implement them into performance tools.

Bibliography

[1] K. Al-Tawil and C. A. Moritz, "Performance modeling and evaluation of mpi," *Journal of Parallel and Distributed Computing*, vol. 61, no. 2, pp. 202–223, 2001.

[2] K. London, S. Moore, P. Mucci, K. Seymour, and R. Luczak, "The papi cross-platform interface to hardware performance counters," 06 2021.

[3] W. Jalby, C. Valensi, M. Tribalat, K. Camus, Y. Lebras, E. Oseret, and S. Ibnamar, "One view: A fully automatic method for aggregating key performance metrics and providing users with a synthetic view of hpc applications," in *Tools for High Performance Computing 2018 / 2019* (H. Mix, C. Niethammer, H. Zhou, W. E. Nagel, and M. M. Resch, eds.), (Cham), pp. 219–235, Springer International Publishing, 2021.

[4] R. B. Roy, T. Patel, V. Gadepally, and D. Tiwari, *Bliss: Auto-Tuning Complex Applications Using a Pool of Diverse Lightweight Learning Models*, p. 1280–1295. New York, NY, USA: Association for Computing Machinery, 2021.

[5] B. Aksar, Y. Zhang, E. Ates, B. Schwaller, O. Aaziz, V. J. Leung, J. Brandt, M. Egele, and A. K. Coskun, "Proctor: A semi-supervised performance anomaly diagnosis framework for production hpc systems," in *High Performance Computing* (B. L. Chamberlain, A.-L. Varbanescu, H. Ltaief, and P. Luszczek, eds.), (Cham), pp. 195–214, Springer International Publishing, 2021.

[6] J. Filipovic, J. Hozzová, A. Nezarat, J. Olha, and F. Petrovič, "Using hardware performance counters to speed up autotuning convergence on gpus," 02 2021.

[7] H. Sanders, "Garbage in, garbage out: How purportedly great ml models can be screwed up by bad data," in *Proceedings of Blackhat*, 2017.

[8] R. S. Geiger, K. Yu, Y. Yang, M. Dai, J. Qiu, R. Tang, and J. Huang, "Garbage in, garbage out? do machine learning application papers in social computing report where human-labeled training data comes from?," in *Proceedings of the 2020 Conference on Fairness, Accountability, and Transparency*, FAT* '20, (New York, NY, USA), p. 325–336, Association for Computing Machinery, 2020.

[9] L. T. R. EdD and K. W. F. PhD, "Garbage in, garbage out: Having useful data is everything," *Measurement: Interdisciplinary Research and Perspectives*, vol. 9, no. 4, pp. 222–226, 2011.

[10] L. Adhianto and B. Chapman, "Performance modeling of communication and computation in hybrid mpi and openmp applications," *Simulation Modelling Practice and Theory*, vol. 15, no. 4, pp. 481–491, 2007. Performance Modelling and Analysis of Communication Systems.

[11] C. Allande, J. Jorba, A. Sikora, and E. César, "A performance model for openmp memory bound applications in multisocket systems," in *Proceedings of the International Conference on Computational Science, ICCS 2014, Cairns, Queensland, Australia, 10-12 June, 2014* (D. Abramson, M. Lees, V. V. Krzhizhanovskaya, J. J. Dongarra, and P. M. A. Sloot, eds.), vol. 29 of *Procedia Computer Science*, pp. 2208–2218, Elsevier, 2014.

[12] C. Allande, J. Jorba, A. Sikora, E. César, and G. al, *Performance model based on memory footprint for OpenMP memory bound applications*, vol. 27 of *Advances in Parallel Computing*, pp. 73–82. 1 ed., Jan. 2016.

[13] C. Rosas, A. Sikora (Morajko), J. Jorba Esteve, and E. César, "Workload balancing methodology for data-intensive applications with divisible load," pp. 48–55, 10 2011.

[14] A. Moreno, E. César, A. Guevara, J. Sorribes, T. Margalef, and E. Luque, "Dynamic pipeline mapping (dpm)," pp. 295–304, 01 1970.

[15] A. Moreno, A. Sikora (Morajko), E. César, J. Sorribes, and T. Margalef, "Hedpm: load balancing of linear pipeline applications on heterogeneous systems," *The Journal of Supercomputing*, vol. 73, pp. 1–23, 09 2017.

[16] A. Morajko, E. César, P. Caymes-Scutari, T. Margalef, J. Sorribes, and E. Luque, "Automatic tuning of master/worker applications," in *Proceedings of the 11th International Euro-Par Conference on Parallel Processing*, Euro-Par'05, (Berlin, Heidelberg), p. 95–103, Springer-Verlag, 2005.

[17] A. Morajko, O. Morajko, T. Margalef, and E. Luque, "Mate: Dynamic performance tuning environment," in *European Conference on Parallel Processing*, pp. 98–107, Springer, 2004.

[18] S. Hong and H. Kim, "An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness," *SIGARCH Comput. Archit. News*, vol. 37, p. 152–163, June 2009.

[19] S. Hong and H. Kim, "An integrated gpu power and performance model," *SIGARCH Comput. Archit. News*, vol. 38, p. 280–289, June 2010.

[20] B. Welton and B. P. Miller, "Diogenes: Looking for an honest cpu/gpu performance measurement tool," in *Proceedings of the International Conference for High Performance*

Computing, Networking, Storage and Analysis, SC '19, (New York, NY, USA), Association for Computing Machinery, 2019.

[21] B. Welton and B. P. Miller, "Identifying and (automatically) remedying performance problems in cpu/gpu applications," in *Proceedings of the 34th ACM International Conference on Supercomputing*, ICS '20, (New York, NY, USA), Association for Computing Machinery, 2020.

[22] C. Tapus, I.-H. Chung, and J. Hollingsworth, "Active harmony: Towards automated performance tuning," in *SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pp. 44–44, 2002.

[23] J. A. Nelder and R. Mead, "A Simplex Method for Function Minimization," *The Computer Journal*, vol. 7, pp. 308–313, 01 1965.

[24] R. Miceli, G. Civario, A. Sikora, E. César, M. Gerndt, H. Haitof, C. Navarrete, S. Benkner, M. Sandrieser, L. Morin, and F. Bodin, "Autotune: A plugin-driven approach to the automatic tuning of parallel applications," in *Applied Parallel and Scientific Computing* (P. Manninen and P. Öster, eds.), (Berlin, Heidelberg), pp. 328–342, Springer Berlin Heidelberg, 2013.

[25] R. Miceli, G. Civario, A. Sikora, E. César, M. Gerndt, H. Haitof, C. Navarrete, S. Benkner, M. Sandrieser, L. Morin, F. Bodin, and P. Pekka Manninen, *AutoTune: A Plugin-Driven Approach to the Automatic Tuning of Parallel Applications*, vol. 7782, pp. 328–342. 1 ed., Jan. 2013.

[26] P. G. Kjeldsberg, A. Gocht, M. Gerndt, L. Riha, J. Schuchart, and U. S. Mian, "Readex: Linking two ends of the computing continuum to improve energy-efficiency in dynamic applications," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pp. 109–114, 2017.

[27] J. Schuchart, M. Gerndt, P. G. Kjeldsberg, M. Lysaght, D. Horák, L. Říha, A. Gocht, M. Sourouri, M. Kumaraswamy, A. Chowdhury, M. Jahre, K. Diethelm, O. Bouizi, U. S. Mian, J. Kružík, R. Sojka, M. Beseda, V. Kannan, Z. Bendifallah, D. Hackenberg, and W. E. Nagel, "The readex formalism for automatic tuning for energy efficiency," *Computing*, 2017. doi: `10.1007/s00607-016-0532-7`.

[28] V. Nikl, L. Říha, O. Vysocký, and J. Zapletal, "Optimal hardware parameters prediction for best energy-to-solution of sparse matrix operations using machine learning techniques," in *INFOCOMP 2018*, The Eighth International Conference on Advanced Communications and Computation, pp. 43–48, International Academy, Research, and Industry Association, 2018.

[29] A. Qawasmeh, A. M. Malik, and B. M. Chapman, “Adaptive openmp task scheduling using runtime apis and machine learning,” in *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, pp. 889–895, 2015.

[30] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade, “Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp,” in *2009 International Conference on Parallel Processing*, pp. 124–131, 2009.

[31] P. I. Frazier, “Bayesian optimization,” in *Recent Advances in Optimization and Modeling of Contemporary Problems*, pp. 255–278, INFORMS, 2018.

[32] M. Popov, C. Akel, Y. Chatelain, W. Jalby, and P. de Oliveira Castro, “Piecewise holistic autotuning of parallel programs with cere,” *Concurrency and Computation: Practice and Experience*, vol. 29, 06 2017.

[33] G. Tournavitis, Z. Wang, B. Franke, and M. F. O’Boyle, “Towards a holistic approach to auto-parallelization: Integrating profile-driven parallelism detection and machine-learning based mapping,” in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’09, (New York, NY, USA), p. 177–187, Association for Computing Machinery, 2009.

[34] Z. Wang, G. Tournavitis, B. Franke, and M. F. P. O’boyle, “Integrating profile-driven parallelism detection and machine-learning-based mapping,” *ACM Trans. Archit. Code Optim.*, vol. 11, Feb. 2014.

[35] Z. Wang and M. F. O’Boyle, “Mapping parallelism to multi-cores: A machine learning based approach,” *SIGPLAN Not.*, vol. 44, p. 75–84, Feb. 2009.

[36] A. Collins, C. Fensch, H. Leather, and M. Cole, “Masif: Machine learning guided auto-tuning of parallel skeletons,” in *20th Annual International Conference on High Performance Computing*, pp. 186–195, 2013.

[37] A. Morajko, O. Morajko, T. Margalef, and E. Luque, “Mate: Dynamic performance tuning environment,” in *Euro-Par 2004 Parallel Processing* (M. Danelutto, M. Vanneschi, and D. Laforenza, eds.), (Berlin, Heidelberg), pp. 98–107, Springer Berlin Heidelberg, 2004.

[38] A. Morajko *et al.*, *Dynamic Tuning of Parallel/Distributed Applications.* PhD thesis, Universitat Autònoma de Barcelona.

[39] G. Ravipati, A. R. Bernat, N. E. Rosenblum, B. P. Miller, and J. K. Hollingsworth, “Toward the deconstruction of dyninst,” 2007.

[40] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall, “The paradyn parallel performance measurement tool,” *Computer*, vol. 28, no. 11, pp. 37–46, 1995.

[41] V. Pillet, J. Labarta, T. Cortes, and S. Girona, “Paraver: A tool to visualize and analyze parallel code,” *WoTUG-18*, vol. 44, 03 1995.

[42] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr, “The scalasca performance toolset architecture,” *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 702–719, 2010.

[43] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel, “The vampir performance analysis tool-set,” in *Tools for High Performance Computing* (M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, eds.), (Berlin, Heidelberg), pp. 139–155, Springer Berlin Heidelberg, 2008.

[44] A. Sikora (Morajko), P. Caymes-Scutari, T. Margalef, and E. Luque, “Mate: Monitoring, analysis and tuning environment for parallel/distributed applications,” *Concurrency and Computation: Practice and Experience*, vol. 19, pp. 1517 – 1531, 08 2007.

[45] A. Morajko, *Dynamic tuning of parallel/distributed applications.* Universitat Autònoma de Barcelona,, 2004.

[46] OpenMP Architecture Review Board, “Openmp application programming interface 4.0,” tech. rep., 2013.

[47] A. Martínez, A. Sikora, E. César, and J. Sorribes, “Elastic: A large scale dynamic tuning environment,” *Sci. Program.*, vol. 22, p. 261–271, Oct. 2014.

[48] Z. Li, A. Jannesari, and F. Wolf, “Discovery of potential parallelism in sequential programs,” in *2013 42nd International Conference on Parallel Processing*, pp. 1004–1013, 2013.

[49] M. Gerndt, K. Fürlinger, and E. Kereku, “Periscope: Advanced techniques for performance analysis.,” pp. 15–26, 01 2005.

[50] R. Mijaković, M. Firbach, and M. Gerndt, “An architecture for flexible auto-tuning: The periscope tuning framework 2.0,” in *2016 2nd International Conference on Green High Performance Computing (ICGHPC)*, pp. 1–9, 2016.

[51] J. Filipovič, F. Petrovič, and S. Benkner, “Autotuning of opencl kernels with global optimizations,” in *Proceedings of the 1st Workshop on AutotuniNg and ADaptivity AppRoaches for Energy Efficient HPC Systems*, ANDARE ’17, (New York, NY, USA), Association for Computing Machinery, 2017.

[52] S. S. Shende and A. D. Malony, “The tau parallel performance system,” *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[53] C. Wood, S. Sane, D. Ellsworth, A. Gimenez, K. Huck, T. Gamblin, and A. Malony, “A scalable observation system for introspection and in situ analytics,” in *Proceedings of the*

5th Workshop on Extreme-Scale Programming Tools, ESPT '16, p. 42–49, IEEE Press, 2016.

[54] L. Pouchard, K. Huck, G. Matyasfalvi, D. Tao, L. Tang, H. V. Dam, and S. Yoo, "Prescriptive provenance for streaming analysis of workflows at scale," in *2018 New York Scientific Data Summit (NYSDS)*, pp. 1–6, 2018.

[55] G. Hackeling, *Mastering Machine Learning With Scikit-Learn.* Packt Publishing, 2014.

[56] A. Burkov, *The Hundred-Page Machine Learning Book.* Andriy Burkov, 2019.

[57] D. T. Pham, S. S. Dimov, and C. D. Nguyen, "Selection of k in k-means clustering," *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science*, vol. 219, no. 1, pp. 103–119, 2005.

[58] L. Hansen and J. Larsen, "Unsupervised learning and generalization," in *Proceedings of International Conference on Neural Networks (ICNN'96)*, vol. 1, pp. 25–30 vol.1, 1996.

[59] I. Jolliffe, *Principal Component Analysis*, pp. 1094–1096. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

[60] A. Ozgur, "Supervised and unsupervised machine learning techniques for text document categorization," 01 2004.

[61] H. A. Ismail, "Learning data science: Day 10 - classification, k-nearest neighbors, and cross validation." `https://haydar-ai.medium.com/learning-data-science-day-10-classification-k-nearest-neighbors-and-cross-validation-d7d58dbe1fed`, Jan 2017. Accessed: 10-09-2021.

[62] S. R. Safavian and D. Landgrebe, "A survey of decision tree classifier methodology," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 21, no. 3, pp. 660–674, 1991.

[63] V. Kecman, *Support Vector Machines – An Introduction*, vol. 177, pp. 605–605. 05 2005.

[64] S. Haykin, *Neural Networks: A Comprehensive Foundation.* Prentice Hall, 1999.

[65] S. Sharma, "Activation functions in neural networks," *International Jounal of Engineering Applied Sciencies and Technology*, vol. 4, 2020.

[66] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall, "Activation functions: Comparison of trends in practice and research for deep learning," *CoRR*, vol. abs/1811.03378, 2018.

[67] A. J. Bishara and J. B. Hittner, "Testing the significance of a correlation with nonnormal data: comparison of pearson, spearman, transformation, and resampling approaches.," *Psychological methods*, vol. 17, no. 3, p. 399, 2012.

[68] J. Alcaraz, A. Sikora, and E. César, "Dynamic tuning of openmp memory bound applications in multisocket systems using MATE," in *The 47th International Conference on Parallel Processing, ICPP 2018, Workshop Proceedings, Eugene, OR, USA, August 13-16, 2018*, pp. 37:1–37:10, ACM, 2018.

[69] J. Alcaraz, A. Sikora, and E. Cesar, "Dynamic tuning of openmp memory bound applications in multisocket systems using mate," in *Proceedings of the 47th International Conference on Parallel Processing Companion*, ICPP '18, (New York, NY, USA), Association for Computing Machinery, 2018.

[70] J. Tang, S. Alelyani, and H. Liu, *Feature selection for classification: A review*, pp. 37–64. CRC Press, Jan. 2014.

[71] D. M. Hawkins, "The problem of overfitting," *Journal of Chemical Information and Computer Sciences*, vol. 44, no. 1, pp. 1–12, 2004. PMID: 14741005.

[72] L. Yu and H. Liu, "Efficient feature selection via analysis of relevance and redundancy," *Journal of Machine Learning Research*, vol. 5, pp. 1205–1224, Oct. 2004. Publisher Copyright: © 2004 Lei Yu and Huan Liu.

[73] J. D. MCCALPIN, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, 1995.

[74] J. Alcaraz, A. Sikora, and E. César, "Hardware counters' space reduction for code region characterization," in *Euro-Par 2019: Parallel Processing* (R. Yahyapour, ed.), (Cham), pp. 74–86, Springer International Publishing, 2019.

[75] G. H. Nguyen, A. Bouzerdoum, and S. L. Phung, "Learning pattern classification tasks with imbalanced data sets," 2009.

[76] A. Nath and K. Subbiah, "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*, vol. 272, pp. 294–305, 2018.

[77] M. M. Rahman and D. N. Davis, "Addressing the class imbalance problem in medical datasets," *International Journal of Machine Learning and Computing*, vol. 3, pp. 224–228, 2013.

[78] L. Jäntschi and S.-D. Bolboaca, "Pearson versus spearman, kendall's tau correlation analysis on structure-activity relationships of biologic active compounds," *Leonardo Electronic Journal of Practices and Technologies*, vol. 6, pp. 76–98, 2005.

[79] M. M. Mukaka, "A guide to appropriate use of correlation coefficient in medical research," *Malawi medical journal*, vol. 24, no. 3, pp. 69–71, 2012.

[80] J. D. McCalpin, “Stream: Sustainable memory bandwidth in high performance computers,” *Link: www.cs.virginia.edu/stream/*, 1995.

[81] T. Yuki, “Understanding polybench/c 3.2 kernels,” in *International workshop on Polyhedral Compilation Techniques (IMPACT)*, pp. 1–5, 2014.

[82] T. Yuki and L.-N. Pouchet, “Polybench 4.0,” 2015. Accessed: April 21 2020.

[83] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter, “Self-normalizing neural networks,” in *Advances in neural information processing systems*, pp. 971–980, 2017.

[84] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: a simple way to prevent neural networks from overfitting,” *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[85] D. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *International Conference on Learning Representations*, 12 2014.

[86] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter, “Self-normalizing neural networks,” in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS’17, (Red Hook, NY, USA), p. 972–981, Curran Associates Inc., 2017.

[87] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, *et al.*, “The nas parallel benchmarks,” *The International Journal of Supercomputing Applications*, vol. 5, no. 3, pp. 63–73, 1991.

[88] J. Alcaraz, S. Sleder, A. TehraniJamsaz, A. Sikora, A. Jannesari, J. Sorribes, and E. Cesar, “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, 2021.

[89] C. Phua, D. Alahakoon, and V. Lee, “Minority report in fraud detection: Classification of skewed data,” *SIGKDD Explor. Newsl.*, vol. 6, p. 50–59, June 2004.

[90] S. Bhattacharyya, S. Jha, K. Tharakunnel, and J. C. Westland, “Data mining for credit card fraud: A comparative study,” *Decision Support Systems*, vol. 50, no. 3, pp. 602–613, 2011. On quantitative methods for detection of financial fraud.

[91] M. Zhu, J. Xia, X. Jin, M. Yan, G. Cai, J. Yan, and G. Ning, “Class weights random forest algorithm for processing class imbalanced medical data,” *IEEE Access*, vol. 6, pp. 4641–4652, 2018.

[92] P. Cunningham and S. Delany, *Underestimation Bias and Underfitting in Machine Learning*, pp. 20–31. 04 2021.

[93] S. Kotsiantis, D. Kanellopoulos, and P. Pintelas, "Handling imbalanced datasets: A review," *GESTS ICSSE*, vol. 30, pp. 25–36, 11 2005.

[94] Z. Zheng, Y. Cai, and Y. Li, "Oversampling method for imbalanced classification," *Computing and Informatics*, vol. 34, pp. 1017–1037, 01 2015.

[95] N. Chawla, K. Bowyer, L. Hall, and W. Kegelmeyer, "Smote: Synthetic minority oversampling technique," *J. Artif. Intell. Res. (JAIR)*, vol. 16, pp. 321–357, 06 2002.

[96] G. Kovács, "Smote-variants: A python implementation of 85 minority oversampling techniques," *Neurocomputing*, vol. 366, pp. 352–354, 2019.

[97] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. C. Courville, and Y. Bengio, "Generative adversarial nets," in *NIPS*, pp. 2672–2680, 2014.

[98] A. K. Gangwar and V. Ravi, "Wip: Generative adversarial network for oversampling data in credit card fraud detection," in *Information Systems Security*, 2019.

[99] V. Sampath, I. Maurtua, J. Aguilar, and A. Gutierrez, "A survey on generative adversarial networks for imbalance problems in computer vision tasks," *Journal of Big Data*, vol. 8, 01 2021.

[100] M. Kukar and I. Kononenko, "Cost-sensitive learning with neural networks," in *ECAI*, 1998.

[101] C. Elkan, "The foundations of cost-sensitive learning," *Proceedings of the Seventeenth International Conference on Artificial Intelligence: 4-10 August 2001; Seattle*, vol. 1, 05 2001.

[102] scikit learn, "Estimate class weights for unbalanced datasets.." `https://scikit-learn.org/stable/modules/generated/sklearn.utils.class_weight.compute_class_weight.html`, 2020. Accessed: 10-09-2021.

[103] T. G. Dietterichl, "Ensemble learning," in *The Handbook of Brain Theory and Neural Networks* (M. Arbib, ed.), pp. 405–408, MIT Press, 2002.

[104] A. Lorena, A. Carvalho, and J. Gama, "A review on the combination of binary classifiers in multiclass problems," *Artificial Intelligence Review*, vol. 30, no. 1-4, pp. 19–37, 2008.

[105] G. Biau, "Analysis of a random forests model," *JMLR*, vol. 13, 05 2010.

[106] E. César, A. Moreno, J. Sorribes, and E. Luque, "Modeling master/worker applications for automatic performance tuning," *Parallel Computing*, vol. 32, pp. 568–589, 2006.

[107] H. Jin, Q. Song, and X. Hu, "Auto-keras: An efficient neural architecture search system," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 1946–1956, ACM, 2019.

[108] L. Xu, M. Skoularidou, A. Cuesta-Infante, and K. Veeramachaneni, "Modeling tabular data using conditional gan," in *Advances in Neural Information Processing Systems*, 2019.

[109] N. Ketkar, *Introduction to Keras*, pp. 97–111. Berkeley, CA: Apress, 2017.

[110] L. Breiman, "Random forests," *Machine Learning*, vol. 45, pp. 5–32, Oct 2001.

[111] S. Pennycook, J. Sewall, and V. Lee, "Implications of a metric for performance portability," *Future Generation Computer Systems*, vol. 92, pp. 947–958, 2019.

[112] D. F. Daniel and J. Panetta, "On applying performance portability metrics," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pp. 50–59, 2019.

[113] D. Xu, Y. Shi, I. W. Tsang, Y.-S. Ong, C. Gong, and X. Shen, "Survey on multi-output learning," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 31, no. 7, pp. 2409–2429, 2020.