FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Recommendation Engine for Parallel Loops

José Luís Oliveira da Cunha

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: João Bispo

Co-Supervisor: Jorge Barbosa

October 30, 2020

Recommendation Engine for Parallel Loops

José Luís Oliveira da Cunha

Mestrado Integrado em Engenharia Informática e Computação

October 30, 2020

Abstract

Code parallelization is essential in the present-day to allow software to scale, due to the increase in size and complexity of workloads. In this context, tools that automatically parallelize code have been a focus of research. This is the case since, although parallelization is vital, it is not trivial to manually parallelize code, as it requires a deep knowledge of how the software works and expertise in parallelization techniques.

With the advances in auto-parallelization comes a new problem. Although it is important to detect parallelism in loops, it is also important to detect if said loops should be parallelized. Excessive parallelization, or its application in the wrong loops, can prevent us from taking full advantage of the available parallelism, or even make the final program run slower compared to its sequential counterpart. Thus, it is critical to assess not only what can be parallelized, but also if and how it should be parallelized.

AutoPar is a library for the Clava source-to-source compiler for auto-parallelization of C code. However, it uses a very simple heuristic to determine which loops should be parallelized. This work improves the AutoPar library with state-of-the-art techniques that can classify which loops will provide performance gains when parallelized. This thesis aims at improving this simple heuristic with more sophisticated techniques.

The products of this thesis are a catalog of state-of-art techniques for determining which loops benefit from parallelization and a set of tools that not only helps to evaluate what loops should be parallelized but also serves as a framework for the future users to improve upon them. We expect the benefit from these tools to be the performance improvement (or avoid degradation) obtained in the PolyBench benchmark after using AutoPar when compared with the current heuristic.

Keywords: loop, auto-parallelization, clava, machine learning, compiler, static analysis, dynamic analysis

Acknowledgements

Before beginning this thesis, I must acknowledge and give a special thanks to those who helped and accompanied me along this journey namely:

- Professor João Carlos Viegas Martins Bispo for his supervision, availability, and guidance throughout the thesis development;
- Professor Jorge Manuel Gomes Barbosa for his work as a second supervisor and his help with presentations and the thesis review;
- Professor Ricardo Santos Morla for his help with machine learning concepts;
- The SPeCS Research Group for providing a welcoming environment for the development of this work;
- My parents and my sister for their love and support;
- My friends for always providing the joy and motivation to keep me going.

José Luís Oliveira da Cunha

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Description

For more than a decade, CPU development has focused on multi-core architectures [11, 7]. This means that with each new generation, CPU's have been featuring higher core counts than their predecessors[1]. To extract the maximum performance from these architectures software needs to be able to scale across many cores.

With this necessity comes code parallelization, and more specifically, loop parallelization. Loops tend to be the most important targets of parallelization due to, in general, most of the execution time being spent on them. Parallelizing loops can be an effective method of obtaining performance improvements, but when done manually, usually requires domain-specific expertise and can be a lengthy process.

Auto-parallelization tools automatically detect and apply the necessary changes to the original code to achieve parallelism with little or no external input from the programmer, speeding up the process greatly. However, auto-parallelization tools usually do not achieve the same level of performance of code that has been manually parallelized by experts, and still have room for improvement. AutoPar [2], the automatic parallelization library for the Clava C/C++ source-to-source compiler [4], is one such tool. The library is focused on automatic loop parallelization using OpenMP directives.

1.2 Proposed solution

Contemporary auto-parallelization tools have seen great developments, especially in parallel loop detection. Their rapid development has been a consequence of presenting a more convenient solution for the parallelization process than the conventional manual approach. It is necessary to point out that a good parallelization tool can not only classify what can be parallelized but also what should be. The latter is the problem that we will focus on in this work.

[1] For instance, at the time of writing, the Ryzen Threadripper 3990X has just been launched and contains 64 cores.

When parallelization is applied indiscriminately to all loops that can be parallelized there is a risk of excessive overhead, especially in the case of nested loops. This overhead can compromise the performance gains of the parallelization process and, in some cases, make the program perform worse when compared with the non-parallel version. Clava's AutoPar module is an example of a tool with a far too simplistic heuristic. After detecting what loops are parallelizable, it always parallelizes the outermost parallel loops. This thesis aims at improving AutoPar's heuristic for selecting what loops to parallelize.

Prema et al. [13] show that the process of selecting which loops to parallelize in several auto-parallelization tools can still be improved upon. This is also the case of AutoPar, the tool that will be our target for this work. Currently, AutoPar has the capability of detecting parallel loops and applying the necessary OpenMP directives but lacks a sophisticated method to decide which loops will be changed. As of now, it always adds the pragmas to simple `for` loops, and in the case of nested loops, it only adds them to the outermost parallel loop. This thesis looks to replace this naive approach with a more informed solution that can improve AutoPar's performance (or avoid degradation) in the scenarios described before.

The approach used for this problem was to use machine learning algorithms to predict when a loop should be parallelized. To achieve this goal 3 tools were developed.

The first tool is the ProgramAnalyzer and it is in charge of loading the target source code, analyze it, and extract what loops can be parallelized, their features and pragmas, and their respective speedups. To be able to do this, the tool is coded in a mix of javascript and LARA and makes use of Clava, including the AutoPar module. If the objective is not to build new models but only to make predictions, the speedup calculation can be skipped since running it would defeat the point of trying to predict it.

The second tool is responsible for using the data extracted with the ProgramAnalyzer, processing it, and using it with machine learning algorithms to create models. These models are supposed to be used to later predict what to do with new loops. This tool is coded in Python and makes use of several machine learning and data-oriented modules like scikit-learn, tensorflow, and imblearn. The models, their parameters, and some intermediate outputs are stored in files to inform the user of what is happening in each stage of the process. Like previously, if the objective is to simply make a prediction, all the steps exclusive to model building can be skipped.

The final tool is the ModelTester. This tool is similar to the ProgramAnalyzer in behavior but they try to achieve different goals. Once again it's coded in javascript and LARA and makes use of Clava and AutoPar. The purpose of this tool is to use the predictions made by models built, and apply them to the source code. Each benchmark provided is parallelized 2 times. One time using the predictions from the model and another using the AutoPar's heuristic. Only the loops present inside the designated "kernel" section of the code can be parallelized and this section is timed to compare the performance of both approaches. In the end, it outputs not only the times of each approach but also some statistics to better understand how each approach behaved in each benchmark.

The main contributions of this thesis are:

- Literature revision on the selection of loops for parallelization;
- A set of highly parameterized and extensible tools, that provide an accessible way to make additional tests and create new models;
- A new method for the AutoPar module of Clava to decide which loops to parallelize;

1.3 Structure of the Dissertation

This report is structured in 5 additional chapters. Chapter 2 provides an analysis of the current state-of-the-art parallelization tools and the use of cost models and machine learning for loop parallelization evaluation. Chapter 3 states the problem and presents the developed solution. Chapter 4 goes in-depth in explaining the tools developed for the proposed solution and their inner workings. Chapter 5 presents and analyzes the results of the implementation of the proposed solution. Chapter 6 delivers some conclusions on the results achieved as well as making some assessments on possible ways to improve this work in the future.

Chapter 2

Related Work

This chapter presents the research done on state-of-the-art auto-parallelization tools and techniques to characterize and filter parallel loops. The filtering techniques were further divided into cost models and machine learning approaches.

2.1 Automatic Parallelization Tools

There are several different tools capable of parallelizing code automatically, and we can benefit from understanding the techniques used, and the scenarios in which these tools have shortcomings. Even if the focus of this thesis is not on the parallelization but on deciding which loops to parallelize, some of the knowledge related to the auto parallelization tools might help to find scenarios in which parallelization is applied but shouldn't be.

2.1.1 Auto-parallelization pitfalls

Prema et al. [13] selected several auto parallelization tools (i.e., Cetus, Par4All, Pluto, Parallware, ROSE, ICC) and tested them the NAS Parallel Benchmarks to find pitfalls. They used `gprof` to find the function call that took most of the execution time. The main pitfalls were enumerated and each function was labeled according to the pitfalls found within them. Solutions for each pitfall were also proposed. The results found appear to support the proposition that better parallelization filtering is indeed necessary. One example is in the CG benchmark, the tool Par4All achieves an 11x speedup while ICC only achieves a 2.5x speedup. The reason for this discrepancy is that ICC parallelizes dependant loops leading to excessive overhead. In another example, in the LU benchmark, the roles are reversed. ICC achieves a 5.5x speedup while Par4All achieves a speedup of less than 1x. It is stated that Par4All isn't as efficient in parallelizing these loops which leads to the poor performance recorded.

Even if the difference in speedups of each tool comes down to the techniques used, Prema et al. demonstrate that the benefits of filtering loops for parallelization is two-fold: in the first case,

loop filtering could have improved the performance of the parallel application by preventing the excessive overhead caused by ICC parallelization. As for the second case, loop filtering could have prevented performance from going below that of the sequential version.

2.1.2 AutoPar-Clava

Before analyzing the literature, we'll give a brief overview of what is CLAVA. Clava is a source-to-source compiler for C and C++ code. It is built using the LARA framework which is used to build source-to-source compilers. These compilers use scripts written in LARA, a language built on top of javascript. These scripts allow these compilers to analyze source code and to apply transformations. In the case of AutoPar, parallelizing loops is a product of such transformations and analysis. These transformations are applied to an intermediate representation of the source code. Instead of modifying the code directly, it is translated to an AST formed by several nodes. Each of these nodes is called a join point and represents an element of the source code. When the necessary analysis and transformations are done, the AST is translated to source code again.

Arabnejad et al. [2, 3] present an overview of the inner-workings of the AutoPar library for the Clava compiler [4] as well as comparing its performance against similar tools.

They mention some of the techniques implemented in AutoPar and depict the parallelization algorithm followed by the library. This is relevant for the project at hand since one of the objectives is integrating the developed solution in this library. The results show that the library is already quite competent at handling parallelization in comparison with its contemporaries, but that it still has room for improvement.

More importantly, they mention that they follow a simple algorithm for selecting which of the parallel loops are effectively parallelized, which is to parallelize all loops that are not already inside a parallel loop.

2.2 Cost Models

Cost models can be a viable way to achieve our goal. The main purpose of a cost model is to make estimates on some target given an input. Applying this to source code, more specifically parallel loops, this means estimating the benefits of parallelization based on metrics that can be extracted from the loop. The following articles are related to the applications of cost models in the prediction of parallelism benefits, the creation or extension of cost models, skeleton models, and an overview of performance modeling.

2.2.1 Tolubaeva et al

Tolubaeva et al. [16] created a cost model to detect false sharing based on existing cost models in the Open64 compiler. The Open64 compiler has 3 built-in cost models: the processor model, the cache model, and the parallel model. With these 3 models, they created a new cost model that estimates the number of false sharing in a parallel loop at compile time. This work is relevant

because it shows that new cost models can be built upon existing ones, and it makes estimates on parallel loops at compile time. Although these target of the estimates does not coincide with our goal, it is still an approach to keep in mind.

2.2.2 DSWP

Liu et al. [9] work is directed at Decoupled Software Pipelining Parallelization. In this case, loops are broken into independent parts and subsequently ran in parallel. OpenMP's `task` and `master` directives are used to emulate this parallelism style. The objective of their work is to extend Open64's parallel cost model to support this kind of parallelism. The resulting cost model is then able to predict if this kind of parallelization is beneficial or not. This work presents a solution that can be adapted to our problem. Not only does it deal with cost models and parallel loops, but it shows that the existing models can be extended to make estimates on the benefits of parallelization, just like we intend to.

Zhang et al. [17] work makes use of both machine learning and skeletons for performance prediction. The skeleton is based on traces of program executions and is used to predict the performance of the original program by attempting to replay its behavior. The problem is that skeletons require fixed inputs and so machine learning is used to create a dynamic skeleton. Machine Learning is used to predicted performance based on varying inputs, and the predicted result is fed to the skeleton model. Skeletons can have potential, but they are better suited for application with its parallelism built from skeletons as well. The work presented in this article is applied to MPI-based parallelism with MPI, while our work is based on OpenMP pragmas.

Singleton [15] presents an overview of performance modeling, explains its basics and its relation with performance testing, and compares 3 different techniques. The techniques presented are volumetric analysis, queueing theory, and simulation modeling. The 3 techniques are compared in 4 different dimensions: the amount of data required, speed of initial turnaround of answers, questions that can be answered, and system complexity. The article is not directly related to loops or automatic parallelization but provides an introductory view of performance modeling.

2.3 Machine Learning

Machine Learning is an approach that can be used to implement loop filtering, and when used correctly can have better results than other methods, at the cost of model transparency. Another factor to take into account is that machine learning results greatly depends on the datasets used and on the selected features. The following works are related to parallelism and machine learning, and how they extract knowledge of the models, features, and datasets.

2.3.1 Purnell et al

Purnell et al. [14] propose the use of neural networks to solve the problem of automatic parallelization. Their approach can be summarized in parsing the source-code to a graph representation, applying the necessary transformations on the graph, and finally converting the graph back to source-code, already parallelized. The approach has potential, but there are a few concerns. An apparent flaw is the size of the dataset. It claims the dataset had only 750 loops, which seem to be a very small number for the proper training of a neural network. Another issue is the lack of results. A solution proposal is made, but there are no results to back it up.

2.3.2 Maramzin et al

Maramzin et al. [10] present a machine learning solution for the selection of loops to parallelize and ranks them by their importance. Although this solution is meant to aid manual parallelization, the final product's output is similar to what we need. The differences being that the output ranks loops by the probability of parallelization success and achievable speedup, while we are only concerned with the latter. An important part of this work is the selection of the features to be used. The features are enumerated and accompanied by a brief explanation for their choice. Additionally, they show the ranking of the features' importance based on the model. Finally, they display the solutions' results using various models and detail the reasons for the failure of parallelism detection by the ICC. Once again, the number of loops used, 1415, seems to be smaller than it should for a machine learning approach.

2.3.3 Liu et al

Liu et al. [8] use a machine learning model to predict if it is worth parallelizing a loop with speculative multithreading. The features to represent the loops in the model are collected from static and dynamic analysis and are listed in the article. The model used was a k-NN and it showed small improvements. The loops come from the Olden benchmark suite. Once again, the main takeaway from the work is the selection of features used for the model's creation.

2.3.4 Chen et al

Chen et al. [6] use a different approach in this article. They create several versions of the same parallel code using OpenMP directives and choose the best one at runtime. The difference between the versions is only the thread count, so the scope is rather limited. Several versions are then put inside a switch-case and, one of them is chosen at runtime based on the loop size. The number of versions created is limited to avoid code explosion. These are tested with certain workloads to evaluate their performance and are then ranked based on a point system. The best versions are then selected with the requirement that they must have a certain distance between each other. This is done to prevent the selection of a cluster of similar versions. A k-NN model is used to choose the best version based on loop size. This work presents an interesting approach to solve our problem.

With additional loop features and more variables to configure the OpenMP directives, it can be seen as a possible solution.

2.4 Summary

In the literature review presented in this chapter, we covered research related to the existing tools, cost models, and machine learning. In regards to the tools, we reinforced the need for loop filtering by analyzing the tools pitfalls and received insight on where the final solution will be integrated into the AutoPar parallelization process. For cost models, we observed their potential based on the work done in cost models of the Open64 compiler. There was also an investigation of the use of skeletons and performance modeling in general. For machine learning, we found several promising approaches, and more importantly, good data on features to be used for the creation of these models. One common downside was the size of the datasets used to create the models. The information gathered from the literature gives us insight into multiple ways of approaching the problem and ultimately serves as a starting point for the development of our solution for filtering loops for parallelization.

Chapter 3

Loop Filtering Approach

This chapter presents the proposed solution. We start by presenting a high-level view of the approach, going over the workflow used. Then we detail the expected results and explain how we intend to evaluate and validate them.

3.1 Solution Outline

We propose the use of Clava to analyze source code and extract loop features that are then used as data for the development of a machine learning model capable of selecting which loops to parallelize. After the models are built they can be used to predict whether or not a loop should be parallelized based on the extracted features.

The implementation of this solution, illustrated in diagram 3.1, is divided into 3 main components:

- ProgramAnalyzer - The first component is a LARA script which is responsible for analyzing the provided source code, extract static and dynamic features of loops that are found to be parallelizable, and time them both when they're running in parallel and sequential.
- Machine learning models - The second component is a group of Python scripts that use the data from the previous component to create machine learning models tasked with predicting which loops to parallelize. This component is also responsible for all the necessary data processing prior to the creation of the models. It is also used to make predictions based on previously built models.
- ModelTester - The final component is a LARA script similar to the first component, in which the source code is parallelized based on the AutoPar's current naive method and based on models' predictions from the previous component. The two approaches are then compared in order to assess the performance of the models.

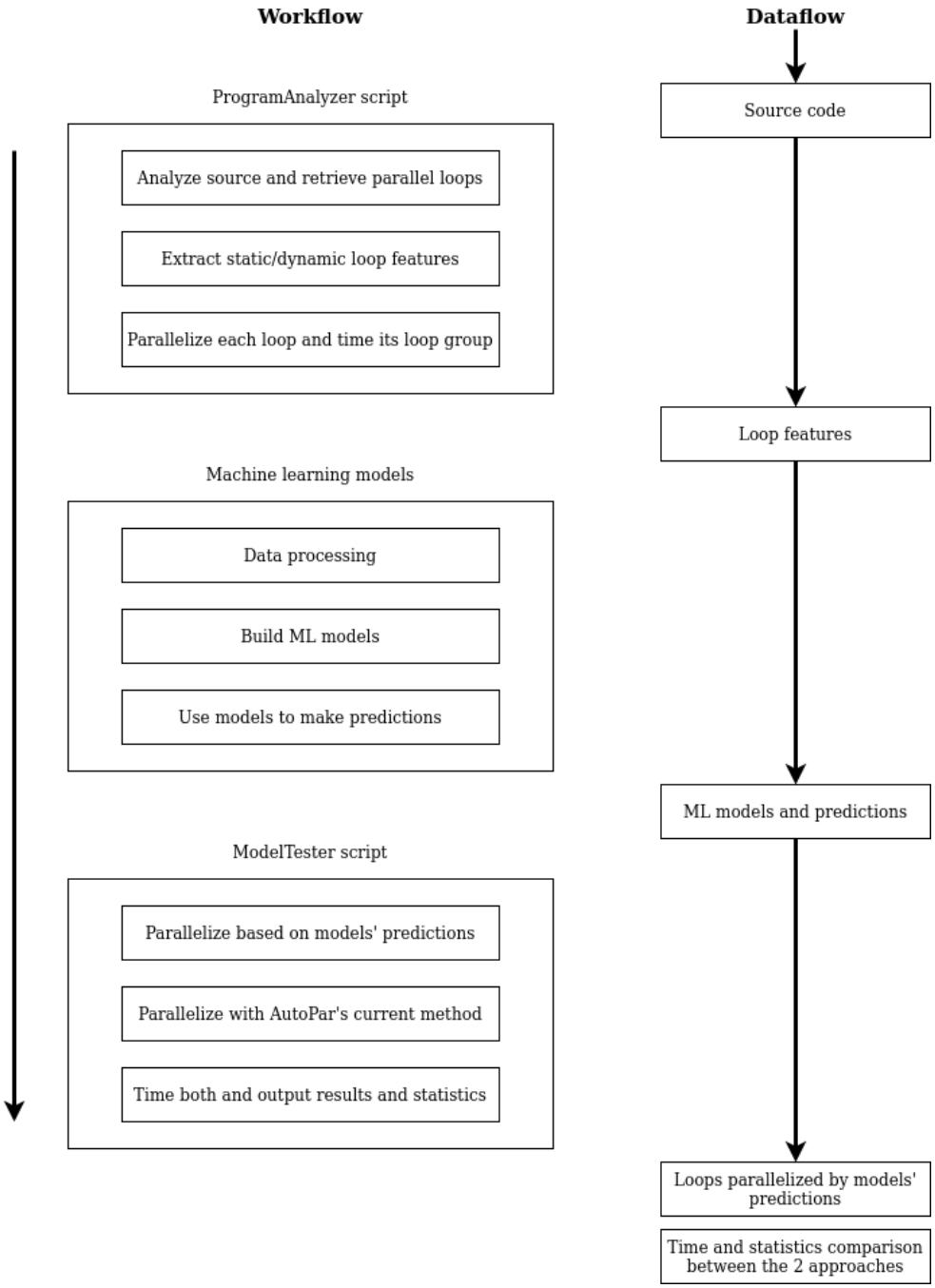

Figure 3.1: Workflow diagram

3.2 Algorithms

Machine Learning is a large field and provides a wide range of algorithms to solve different problems. The k-NN algorithm, which was used by Liu, et al. [8] and Chen, et al. [6] can provide a starting point but work done is not limited to a single algorithm. An important factor in this research is seeing how different algorithms fare in the task of loop selection. This not only provides

more choice for picking the best algorithm but can also serve as a knowledge base for future work in this field.

Deep Learning, a subfield of machine learning has the disadvantage of being less transparent in its inner workings in comparison with other algorithms but, in general, carries the potential for better results as well.

For this purpose, libraries like TensorFlow [1] and scikit-learn [12] for Python not only provide required algorithms but also allow the automation of the model creation and result extraction process. These were the benefits that ultimately made them the choice for this thesis.

From the perspective of machine learning, the problem can be seen as either a classification or a regression. From a classification perspective, there is a binary output for each loop which indicates if it should be parallelized or not. From a regression perspective, there is a numerical output for each loop that estimates the achievable speedup (or slowdown).

Each approach has its advantages and disadvantages. On one hand, classification is expected to yield better results due to the binary nature of the target. On the other hand, although regression models aren't expected to work as well because of their continuous target, they have the possibility of conveying much more information. If two loops have a speedup of 2 and 4 respectively, both models are expected to mark it as loops to classify but the classification output doesn't distinguish between the two, while the regression output makes it clear that even though both achieve speedup above 1, the second loop achieves a significantly higher speedup and is preferred over the first one. Both approaches are tested. This solution takes inspiration from the work of Maramzin, et al. [10], but focuses exclusively on loop selection.

For this thesis the following algorithms from the scikit-learn Python module are tested:

- Regression:
  - SVR.
  - Ridge.
  - DecisionTreeRegressor.
- Classification:
  - C-Support Vector Classification - referred to in this thesis as SVC.
  - KNeighborsClassifier.

Worth keeping in mind that throughout development more models were tested but were eventually discarded due to their poor performance. This helped keep the focus on what appeared to be working to avoid wasting time. The code for these is still present in the current version of the tool. Most work and can be tested and tweaked by future users, while a few are fully up to date to the final specification used for the models. Models that were initially considered but were later dropped from testing are:

- scikit-learn regression models:

  - Ordinary Least Squares.
  - Lasso.
  - Elastic Net.
- Neural networks:
  - scikit-learn's MLPRegressor.
  - Tensorflow Keras' neural networks.

3.3 Data Gathering

In Machine Learning, data is a very important aspect of the entire process. It is necessary to have data in both quantity and quality. For this work, there are 3 main concerns with regards to data:

- **Features used to represent loops** - For the models to make good decisions on what loops to filter, the loops in the training set need to be accurately represented. This puts great emphasis on the quality of the features used to categorize each loop. The research made by Liu, et al. [8] provides insight on relevant static and dynamic loop features. The work made by Maramzin, et al. [10] also does this, on top of ranking the features by order of importance, as used by their model.

  In the case of this thesis both static and dynamic features are extracted from the source code. This is done by making use of the Clava functionalities in terms of code analysis and its representation of said code in an Abstract Syntax Tree (AST). The features in question are listed in the tables 4.3, 4.4 and 4.5.

- **Dataset of loops** - The dataset of loops needs not only to be representative of real-world workloads but also needs to be extensive enough to allow the creation and training of proper models. The small size of the dataset was seen as one of the shortcomings of Maramzin, et al work [10]. In the case of this thesis, 3 main benchmarks will be used: Polybench 4.2, NAS, and TEXAS_42. The first two will be used for both training and testing while the latter will only be used for testing. Polybench and NAS were chosen for several reasons:
  - Code is very susceptible to parallelization.
  - Big number of loops to extract features from.
  - They are the main benchmarks used when comparing tools in the field of loop parallelization. This point makes it easy not only to compare with the current AutoPar method but also for others to compare their results with our own.

  Worth mentioning that the NAS and TEXAS_42 benchmarks required a few changes:

  - A single `#pragma kernel` was added in all benchmarks in both benchmark sets to the region of code that was considered to be the kernel of the benchmark, to delimit

where parallelization can occur, and where time is measured for the final result evaluation. Polybench did not require this change because its kernels are already well delimited by calls to functions whose name has the prefix `kernel_`.

- The UA benchmark from the NAS benchmark set has had a variable change its name. The variable "time" of the type "double" was renamed to "time_no_collision". This was necessary due to a name collision when including the header "time.h" that is necessary for several stages.

- **Loop labeling** - Machine Learning methods require labeled data which, in this case, consists of the loops having the corresponding speedup of their parallel version. The Clava C/C++ compiler[1] will be used to acquire this data. Thanks to the use of LARA scripts to analyze and modify the C/C++ code, it is possible to automate the process of compiling and executing each parallel loop in isolation.

  It's important to keep in mind that in the case of nested loops, the parallelization of one affects the performance of the others. Due to this, just timing each loop would not give representative results, as all loops individually could be suitable to be parallelized but coupled together could result in performance degradation. To attempt to avoid this problem loop groups were defined. A loop group is a group of nested loops. The solution is still parallelizing each loop individually to measure their parallel performance but instead of timing only the loop, the timer is set to time the entire loop group. By timing the entire loop group, the timer can capture possible overheads that would "spill" to the parent loops that by just timing the parallel loop in question would not be measured.

3.4 Summary

The proposed solution is to use machine learning to create a model capable of deciding whether or not parallelization is beneficial. This solution is similar to and inspired by the work of Maramzin, et al. [10] and Liu, et al. [8]. The plan is to test several machine learning algorithms and compare their performance. Clava will be used to gather the data from loops and, in the end, to test and compare the new and old approaches.

[1] https://github.com/specs-feup/clava/

Chapter 4

Loop Filtering Tools

This chapter presents the tool flow developed during this thesis and explains its structure and functionalities. It is responsible for extracting features from selected loops, create machine learning models based on those features, and finally test the results of the models against the current solution. The tool flow has 3 main parts, that correspond to the components presented in Section 3.1, two implemented in Lara as Clava libraries (ProgramAnalyser and ModelTester), and one written in Python (Machine Learning Models). Each is discussed in their own section, following their usage order. It is important to state that the sections in this chapter focus on the main steps of the processes. This means that some trivial operations might only be briefly mentioned or outright omitted. This is done to emphasize the main features of the tool flow. For more details, it is advised to check the source code.

4.1 Clava

This section presents some important information that is common to the ProgramAnalyzer and ModelTester scripts since they both use Clava.

4.1.1 Requirements

The scripts require Clava to be installed. The install location is irrelevant, as long as it can be called from the command line. To initiate the scripts the following command is needed. Note that the shell should be in the folder of the file "launcher.config".

clava -c launcher.config

This command calls Clava with the config file specified. The config file used is available alongside the source code.

4.1.2 Folder structure

The script expects a specific folder structure. As said previously, the shell should be in the folder of the config file when launching the scripts. A few folders are expected to be present after the execution ends. These would be:

- cache - contains files cached from every run of the ProgramAnalyzer script. The folder structure inside it mimics the source code structure to facilitate navigation. The files are stored inside the leaf folders. Inside them, there will be 5 files (5 per each distinct groupLoopSize used).
- extraIncludes - contains files necessary for dynamic feature extraction. These .c and .h files are required for counting loop iterations.
- model-outputs - contains the outputs of the machine learning models. These can be selected to be used to evaluate their performance vs the AutoPar naive model.
- model-performance - contains the results of the comparison of the model performance vs the AutoPar naive method.
- outputs - contains the processed source code and binaries of every version generated from the source code provided. The folder structure mimics the source code structure to facilitate navigation.
- params - contains the script main parameters as well as the various run configurations.
- results - contains the results from the ProgramAnalyzer script. These have the feature information of the analyzed source code, and if set, the timings of each parallel version vs the sequential counterpart.
- scripts - contains the LARA scripts.
- sources - contains the various source codes to be used.

Some of these folders are expected to be created by the scripts upon execution. From this list, the mandatory folders that need to be present before execution are: extraIncludes, params, scripts, sources.

4.1.3 Parameter files

The script makes use of several parameter files. This design choice was made early on, not only to make development faster and easier but also to improve future usability. The objective was that future uses of the script shouldn't require code changes, with the exceptions of eventual bug fixes or possibly adding new features. By providing a well-defined parameter interface through these files, other users can run the scripts on their codebases with their desired settings without having to directly modify the source code of the tool. With this approach, we provide customization while

also preventing user errors and decreasing the time necessary to learn how to use this tool. All parameters are in JSON files which also ensures that they are easily readable and editable by both humans and machines.

The script makes use of two types of parameter files, the main parameter file, and the benchmarks parameter files. The main parameter file controls what the script will do for all benchmarks. The benchmark parameter files have parameters related to each benchmark/program.

The main parameter file is always named "mainParams.json" and contains the fields that indicate to the Launcher script what files to load for the ProgramAnalyzer and the ModelTester scripts. These files contain information about the source code to be used as well as optional parameters for the scripts. In the case of the ModelTester script, it also indicates what results of the models should be tested. There is also an option to display a detailed explanation of each parameter.

The second type of parameter file is related to a codebase or a benchmark set. In these files, multiple fields give compilation instructions, like necessary libraries or flags, as well as some options specific to the scripts. The latter can change how some stages behave or outright disable them. The description of these parameters is also displayed when using the option mentioned previously.

4.2 Clava - ProgramAnalyzer Script

This section details how the ProgramAnalyzer script operates, explaining its inputs, outputs, and process of obtaining the latter from the former. The figure 4.1 depicts a high level view of the process. It can be broken down into a few different stages:

1. Benchmark/program files search - The given source folder is searched for source files that are then stored in an object. A different object is created for each benchmark if the source folder represents a benchmark set. If it is a single program a single object is created to store all info.

2. Parallel loop extraction - The source code's for loops are analyzed in order to retrieve the ones that can be parallelized, alongside their respective omp pragmas.

3. Pragma testing - The pragmas retrieved are tested 1 by 1 to make sure that the final code can run successfully if these pragmas are applied. This stage is optional and supports different strictness levels.

4. Parallel loop grouping - The parallel loops are grouped according to their loopGroup. It returns all possible combinations of all the found loopGroups.

5. Feature extraction - The features of each parallel loop are extracted. Can extract both dynamic and static or only static. When extracting dynamic features, it requires counting iterations of all the benchmark/program's loops.

6. Caching - All the information extracted until this stage that is expected to not change from run to run is cached to multiple files in the "cache" folder. If desired, the user can skip extracting all the values again and simply load them from a previously built cache, therefore skipping the previous stages.

7. Time measurements - With all the necessary information extracted/loaded, the script applies the pragmas according to every combination of the loopGroups from the "Parallel loop grouping" stage. Timers are inserted alongside the pragmas to time the execution time of each loopGroup in all the possible combinations. Afterward, each loopGroup is timed when the benchmark/program is running purely sequentially.

8. Output - Finally, after having all the measures and all the extract features, this info is compiled into a single JSON file that is stored in the "results" folder.

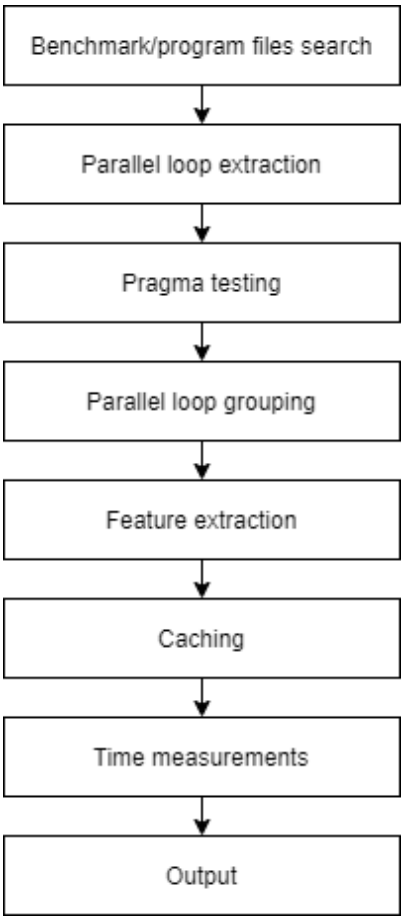

Figure 4.1: ProgramAnalyzer stages

4.2.1 Benchmark/program files search

This is the first stage and it is responsible for searching for the source files inside the designated source folder that contains the benchmarks or program. This stage takes into account the value of the "isBenchmarkSet" parameter in order to choose the mode of operation for the search. It also makes use of the "excludedFiles" and "excludedFolders" parameters. These can be useful when it is desirable to filter out of the search certain files or folders. The file extensions that are searched for are ".c", ".h", ".cpp" and ".hpp". Any file that does not have one of the listed file extensions is not considered a source file and therefore ignored.

When searching for the files, an object is created to store the information related to each benchmark/program. This object stores information related to the necessary files and the folders where data generated by the script will be stored.

If the parameter "isBenchmarkSet" is set to true, the source folder is treated like a folder with several benchmarks/programs inside it. It is important to note that when in this mode of operation, only leaf folders will be considered as benchmark folders. This means that source files that are in non-leaf folders are ignored. Also, each folder is only expected to contain files for a single benchmark although the benchmark can be divided into multiple files. In this mode of operation, the script also takes into account the "benchmarkFolders" parameter. This parameter filters the folders that are searched, ignoring the folders that are not present in this list. Keep in mind that the descendants of the listed folders are still searched. This can be useful in benchmark sets as they are sometimes organized in categories, making it simple to choose to analyze only some categories if so desired.

If the parameter "isBenchmarkSet" is set to false, the source folder is treated as a single program. This means that all source files on this folder and all its descendants are used. Of course, in this case, the source folder and its descendants should contain source files relating to a single program.

4.2.2 Parallel loop extraction

This stage is responsible for analyzing the source code provided and search for parallelizable for loops. Depending on the parameters used it can be done in 3 different ways: parallelize all loops with AutoPar, extract pragmas from a file with the loops already parallelized or read the cached info.

4.2.2.1 Parallelize all loops with AutoPar

This method is more reliable but also more time-consuming. It uses AutoPar functions to determine what loops can be parallelized and extracts the omp pragmas of all the parallelizable loops. To do this, a very computationally expensive analysis of the code must be made. This analysis makes it the slowest method of the 3 but also the most reliable since the results are guaranteed to come from the target source code and it can always be used since it does not require any extra files.

This method is used if the benchmark parameters have "autoParSelectorFlags.readCache" set to False and the current benchmark folder is not listed in the "foldersToGetExpectedC".

4.2.2.2 Extract pragmas from a file

This is the 2nd fastest method. It searches for omp pragmas in the version that has been previously parallelized. The loops are then matched with the source's loops based on their id, and if they exist on both files, the loop's pragma and id are saved. At the end of the process, we have the ids of all parallel loops and their respective pragmas. Since part of the id of a loop comes from the file name, a function was created that converts the loop ids from the expected output file to the original file ids.

When the conditions are met, this approach is very useful for saving time. Since the code does not need to be analyzed to determine what can be parallelized the process is much faster. The downsides are the number of conditions that need to be met for this method to be usable:

- Pre-existing file with all loops parallelized with their respective omp pragmas.
- Support for only 1 source file per benchmark. This limitation is due to the use of the file name for the loop id. To be able to make the id conversion mentioned above, both the original source and the parallelized version can only be a single source file. The previously parallelized version must be named "expected_output.c" and placed in the same folder as the original source file.
- When testing multiple problem sizes for a benchmark, this method cannot be used if there is a reduction on an array whose size is dependant on the problem size flag. This is because the reduction in the pragma will have values related to the array size and this causes problems when the problem size changes and the array size changes as well. The array size is different but the pragma in the expected output file is the same. The mismatch in array sizes will make it impossible to parallelize this loop in any other problem size other than the one used in the expected output file.

This method is used if the benchmark parameters have "autoParSelectorFlags.readCache" set to False and the current benchmark folder is listed in the "foldersToGetExpectedC".

4.2.2.3 Read cached info

This is the fastest method. It simply reads all the information from a cache file from a previous run of the script. The reason for its speed is also its most limiting factor since it requires a previous run of the script with 1 of the methods mentioned previously at 4.2.2.1 and 4.2.2.2. It is important to keep in mind that changes in the target source code or in the tool itself can render previous cache files unusable.

This method is used if the benchmark parameters have "autoParSelectorFlags.readCache" set to True.

4.2.3 Pragma testing

After the parallel loop are determined and their pragmas are known, there is an option to test them. Although this is optional its use is advised. In this stage, the loops retrieved from the previous stage 4.2.2 are parallelized one by one and tested to make sure they work properly. This is important because there is no guarantee that the pragmas retrieved previously work properly with their respective loops. They are supposed to, but no tool is perfect, including AutoPar and since it is likely that these benchmarks will be run multiple times during testing it is best to remove the ones that can be problematic as soon as possible.

As previously mentioned, each pragma is tested in isolation from other omp pragmas. This is done to make sure that if a failure occurs, it is the result of the pragma being tested and not a bad combination of pragmas. Currently, the testing does not support pragma combinations.

The testing strictness level is controlled by the parameter "autoParSelectorFlags.testPragmas". Each level is described in the table 4.1.

Table 4.1: Strictness levels of pragma testing

Parameter value	Description
0	No testing is made and no pragmas are removed.
1	The pragmas are tested individually and the respective loop must finish without errors. An exit statement is added after the loop with the value specified in the parameter "expectedReturnValue". If the return value is the one specified in the parameter it indicates that the loop finished and reached the exit statement successfully. If the return value is different it is considered that an error occurred and the program did not exit properly. This causes the pragma to be removed. Notice that if only the 1st loop execution is tested it is possible that it finishes successfully, but the 2nd execution which would cause an error is not caught. The next strictness level solves this problem.
2	The pragmas are tested individually and the entire benchmark must finish without errors. If the return value is the one specified in the parameter it indicates that the benchmark finished successfully. If the return value is different it is considered that an error occurred and the program did not exit properly. This causes the pragma to be removed. This method is much slower since it runs the entire benchmark with 1 loop parallelized and the rest of the code entirely sequential. On the upside, it is better for loops that are called on multiple points of the benchmark since it tests the complete execution of the loop.
	End of table

4.2.4 Parallel loop grouping

Now that the parallel loops have been selected and the pragmas have been filtered it is time to group them. These are called loop groups and they are formed by an outermost parallel loop and all its nested loops. The outermost loop is also called the main loop since it is the parallel loop of the group with the lowest nesting level, which at the AST level, makes it an ancestor of all the other loops in the group. The grouping is done for 2 main reasons:

- To know which loop pragmas might interfere with one another. Using two nested loops as an example, it is expected that if we parallelize both, there is a chance of a negative performance impact. Another reason to be aware of this kind of situation is that to truly enable parallelization on both loops it is necessary to call the function `omp_set_nested(1)` before that parallel zone. After the parallel zone the same function is called with the argument 0, to disable the flag once again. This is done automatically by the ProgramAnalyzer script.
- Determine which loops are considered main loops. This information will be useful later in 4.2.7.

This stage creates these groups and then returns the subsets of all the groups with a positive size that is smaller or equal to the parameter "loopGroupSizeLimit". These subsets serve as indicators for the later stage 4.2.7 of what combinations of pragmas are to test.

4.2.5 Feature extraction

This is one of the most important stages. Here, the loops previously gathered and filtered are analyzed to have their features extracted. These features are numerical variables that are supposed to represent their respective loop and their characteristics. They can be separated into different categories concerning what they are supposed to represent or how they are extracted.

4.2.5.1 Extracted Features

In chapter 2 we analyzed several solutions, and one recurring task that was deemed essential was what the features should be used to characterize the loops. At first, we considered extracting the same features that were mentioned in the literature but this idea was eventually dropped. The reasons were two-fold:

- Scope - The scope of the thesis did not allow this. The entire project implementation required 3 different tools, each with multiple inner stages, to be able to get the desired end product. Implementing the extraction of the features mentioned in the literature was deemed too costly, taking into account the work required for the rest of the project.
- Clava focus - The purpose of this thesis is not only to improve loop selection for parallelization but more specifically, to improve this process in the AutoPar module of Clava. With this in mind, we decided to make the most use of what information Clava already provided or that could easily be extracted.

The extracted features are either taken directly from the loop join point, extracted while traversing the loop AST or taken from instrumenting and execution of the code.

The names of the extracted features follow specific guidelines and can have one or more prefixes. These prefixes give us the ability to quickly judge the type of feature just by looking at its name. The table 4.2 lists the existing prefixes and their meaning. Dynamic feature extraction can be toggled using the parameter "autoParSelectorFlags.extractDynamicFeatures".

Table 4.2: Loop features' name prefixes

Prefix	**Meaning**
static	Feature can be extracted without executing the code.
dynamic	Feature extraction requires code execution. Features that belong in this category take into account loops iterations. The feature is either itself derived directly from the number of loop iterations or is related to join points. In the case of finding a loop in the body of the loop being analyzed, the join points inside its body and their derived features are multiplied by the calculated loop's average iterations.
instructionInfo	Feature is extracted from the loop AST.
joinpoints	Feature is the number of instances of a specific join point type inside the loop's body. Is a subcategory of "instructionInfo".
joinpointInfo	Feature is a value derived from the join points counted in "joinpoints". Is a subcategory of "instructionInfo".
recursiveJoinpoints	Feature is the number of instances of a specific join point type inside the loop's body and the body of the functions called inside it. If these functions call other functions they will also be searched unless recursion is found. In that case, the call is skipped. If it is dynamic, loops inside the function call also have their body join points multiplied by the calculated loop's average iterations. Is a subcategory of "instructionInfo".
recursiveJoinpointInfo	Feature is a value derived from the join points counted in "recursiveJoinpoints". Is a subcategory of "instructionInfo".
	End of table

A few examples of feature names are:

- static_isMainLoop
- static_instructionInfo_joinpoints_arrayAccess

- static_instructionInfo_recursiveJoinpointInfo_arrayAccessRead
- dynamic_maxIterations
- dynamic_instructionInfo_joinpoints_arrayAccess

Due to the high number of features, their presentation is split into several tables. Some features appear several times in different categories while others are specific to a single category. This is specified for each set of features and the prefixes explained in the table 4.2 should be consulted to understand what each category is.

In the case of static features, in addition to the ones listed in the table 4.3, the categories "joinpoints", "joinpointInfo", "recursiveJoinpoints" and "recursiveJoinpointInfo" are also calculated. The features of these categories are presented in the table 4.5. The left column represents the join point types that are counted and the right column represents the attributes that are extracted from each join point type.

In the case of dynamic features, in addition to the ones listed in the table 4.4, the categories "recursiveJoinpoints" and "recursiveJoinpointInfo" are also calculated. The features of these categories are presented in the table 4.5 as well.

Table 4.3: Static features

Feature	**Description**
isMainLoop	True if the loop is the main loop of the loop group.
nestedLevel	Integer that determines at which nesting level the loop is. A nested level of 0 means the loop has no ancestor loop in the AST. Any loop that is inside the body of another loop has a nested level > 0.
isInnermost	True if the loop is the innermost loop.
isOutermost	True if the loop is the outermost loop.
omp_n_privates	Number of private variables in the omp pragma.
omp_n_first_privates	Number of first private variables in the omp pragma.
omp_n_reductions	Number of reductions in the omp pragma.
omp_n_scalar_reductions	Number of scalar reductions in the omp pragma.
omp_n_array_reductions	Number of array reductions in the omp pragma.
	End of table

Table 4.4: Dynamic features

Feature	Description
maxIterations	Maximum number of iterations of the loop.
minIterations	Minimum number of iterations of the loop.
avgIterations	Average number of iterations of the loop.
stdDevIterations	Standard deviation of the number of iterations of the loop.
timesLoopIsCalled	Number of times the loop is executed. Not to be mistaken with the number of iterations of the loop.
hasIterationVariation	True if the number of iterations of the loop varies.
maxParentIterations	Maximum number of iterations of the parent loop.
minParentIterations	Minimum number of iterations of the parent loop.
avgParentIterations	Average number of iterations of the parent loop.
stdDevParentIterations	Standard deviation of the number of iterations of the parent loop.
hasParentIterationVariation	True if the number of iterations of the parent loop varies.
timesParentLoopIsCalled	Number of times the parent loop is executed. Not to be mistaken with the number of iterations of the parent loop.
	End of table

Table 4.5: joinpoints and recursiveJoinpoints features

Join Points	Attributes
arrayAccess	arrayAccessRead, arrayAccessWrite, arrayAccessReadWrite
binaryOp	binaryOpArithmetic, binaryOpLogic, binaryOpComparison, binaryOpAssignment, binaryOpBitwise, binaryOpBitwiseAssignment, binaryOpArithmeticAssignment
call	callAvgNumberArgs, callExternal, callLevel0, callNested, callFunction, callMethod, callInnerJps, callRecursive
cast	castImplicit, castExplicit
deleteExpr	-
	Continued on next page

Table 4.5 – continued from previous page

Join Points	Attributes
expression	-
if	-
loop	loopFor, loopWhile, loopDoWhile, loopForEach
memberAccess	memberAccessRead, memberAccessWrite, memberAccessReadWrite
memberCall	-
newExpr	-
op	totalArithmetic, totalLogic, totalComparison, totalAssignment, totalBitwise, totalBitwiseAssignment, totalArithmeticAssignment
statement	-
unaryOp	unaryOpArithmetic, unaryOpLogic, unaryOpComparison, unaryOpAssignment, unaryOpBitwise, unaryOpBitwiseAssignment, unaryOpArithmeticAssignment
vardecl	vardeclHasInit
varref	varrefRead, varrefWrite, varrefReadWrite
-	joinpointsTotal
	End of table

4.2.5.2 Counting loop iterations

To be able to extract dynamic features, the script requires information about each loop iterations. To retrieve this information, some code is injected into the original source code. To do this, two extra files are used. They are "autopar_loop_iters.c"[1] and "autopar_loop_iters.h"[2] and they're stored in the "extraIncludes" folder. They contain the definition of a struct, a pointer that is used to dynamically allocate an array, and two functions. One of the functions is to initialize the array and the other is to print the values from the array.

The way this system works is, the array stores structs, each associated with one loop, and they store values that allow the collection of information about the iterations of that specific loop. The benchmark includes these files and calls the initialization function at the start of the benchmark and the print function at the end. Besides this, before each loop, a few lines of code are injected to update the values of each field in the struct related to that loop.

[1] https://github.com/joseloc300/AutoParSelector/blob/v1.1/clava/extraIncludes/autopar_loop_iters.c

[2] https://github.com/joseloc300/AutoParSelector/blob/v1.1/clava/extraIncludes/autopar_loop_iters.h

The counting of loop iterations can be done using two methods and is controlled with the parameter "autoParSelectorFlags.expressionBasedIterCounting". If the parameter is set to true, the injected code uses the expressions in the `for` loop header (i.e., initialization, step, and end condition) to calculate the number of iterations. This is the fastest method and it is also the one we recommend. If the parameter is set to false, the script falls back to count each iteration with a counter, one by one. As expected this method is much slower and should only be used if the previous method does not work.

When benchmark finishes and calls the print function, it prints all the values of all the structs with a set prefix. This prefix is used to identify the loops to whom the values belong. The Lara script then retrieves the console output as a string and parses it to extract all the values.

4.2.6 Caching

After all the previous stages, if the script did not load the values from a cache, it saves all information gathered so far into cache files. This happens automatically and does not require any parameter change. If the script is being run just to generate cache files the parameter "autoParSelectorFlags.onlyCalculateCaches" can be turned on, to skip all steps after the information is cached. In this case, the script will skip to the next benchmark instead of timing the various code versions created. The files are saved in the "cache" folder in a structure similar to the "source" folder structure.

4.2.7 Time measurements

At this point, all the loops features have been extracted and various loops have been divided into groups. Each group represents a version to parallelize and time. To do this a custom version of the `lara.metrics.ExecutionTimeMetric` from the Clava API[3] is used. The developed custom version can measure the time between two specified join points and has the additional feature of inserting an optional `exit()` statement after the first time measurement is made. This addition is valuable because there can be many versions to time per benchmark, due to the number of pragmas and combinations to test, and so we want to optimize the process as much as possible.

The target of the timer is the main loop of the loop group currently being tested. The main loop is measured instead of the parallel loop itself because, as explained earlier, inside a loop group, nested loops can affect the performance of outer loops. To catch the full impact on the performance the main loop is the one timed. This prevents misleading values like a small loop having speedup but causing a slowdown on the outer loop due to overhead. Since all we want to time is the main loop, running the rest of the program is not only unnecessary, since the measures have already been made, but also possibly very slow, since the rest of the program will be sequential. To solve this an `exit()` statement is added after the timer to forcefully terminate the benchmark earlier.

[3] http://specs.fe.up.pt/tools/clava/doc

The measurements are made in nanoseconds and each version can be run multiple times. The number of times each version is run is defined in the parameter "nRuns". There is also the parameter "autoParSelectorFlags.cleanBuilds" that cleans the build folder before each build if set to true. It is advised to keep this turned on, especially if troubleshooting any sort of problem with the builds.

4.2.8 Output

After the "Time measurements" stage is complete, the timed results and all the info related to the loops involved are stored in a "benchResultsObj" object. This object is then added to the "analyzerResultsObj" object.

The stages from "Parallel Loop Extraction" to "Time measurements" are repeated for each benchmark/program found. In the case of benchmarks, if they have flags for different problem sizes set in the "problemSizeFlags" parameter, each benchmark will also be run with every flag once.

After all the runs of all the benchmarks/programs are done the "analyzerResultsObj" object is output to a JSON file named after the "benchGroupName" name param and with a prefix of the time at which the ProgramAnalyzer script started. This file is stored in the "results" folder.

4.3 Machine learning models

In this section, we will go into detail on how the Python scripts operate, describing their inputs and outputs. The figure 4.2 depicts the process in a more high-level view broken down to its main stages:

1. Data loading - Load the outputs of the ProgramAnalyzer script.
2. Data analysis - Calculate the correlation between the features and target. This can be done before and after the data processing.
3. Data processing - Stage responsible for applying modification to the loaded data to improve their quality, before it is fed to the algorithms. These modifications include target processing, handling invalid inputs, scaling the data, feature selection, and using SMOTE for oversampling.
4. Data split - Separation of data into a training and test set.
5. Machine Learning Models - Create the models selected in the parameter file using the loaded data after it is processed and split. The model's hyper-parameters are trained using the tuning files. The best version of the model and its results are stored as files.
6. Predictions and Output - Use the selected model backup to make predictions based on the loaded data. Output the prediction results as well as data relevant to the prediction process.

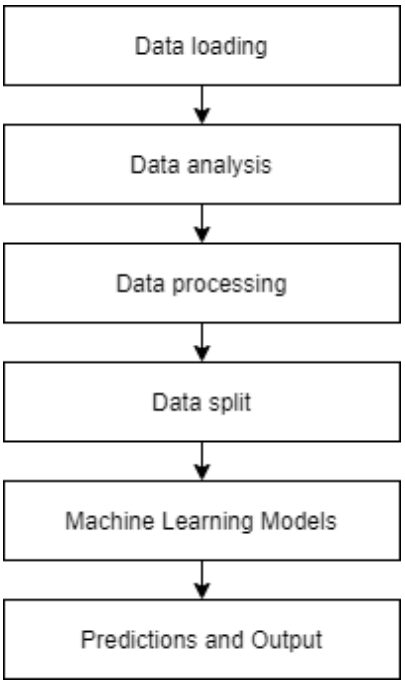

Figure 4.2: Python scripts stages

4.3.1 Folder structure

Like the Clava scripts, the Python scripts expect a specific folder structure::

- data - contains the JSON files output from ProgramAnalyzer script. These files are used as data to create the models.
- models - contains backups of the created models accompanied by their statistics, measured with test data after their creation. The models are divided into two main folders, "classification" and "regression" and each model has its own folder.
- params - contains the file "main_params.json". This file is responsible for configuring all stages and features of the script.
- predictions - contains the JSON files with the predictions made by the selected model. Files are named after the parameter "make_predictions.model_name".
- results - contains JSON and CSV files that are output from some intermediate stages that give insight into what is happening to the data before it is used to create the models.
- tuning - contains the JSON files responsible for the hyper-parameter tuning of the algorithms. They are divided into 2 folders, "classification" and "regression", and are named after their respective algorithm.

4.3.2 Parameter files

The scripts make use of a single parameter file. The decision to do so is the same as stated in 4.1.3 for the Clava scripts. The objective was to make development faster and improve future usability.

The script makes use of a single parameter file. This file is named "main_params.json" and contains parameters to customize all the script stages and even the option to disable some of them.

4.3.3 Data loading

Firstly, this stage loads the "main_params" file, since it contains the parameters that will guide all the functionalities of the scripts. Afterward, it loads all the information from the JSON files in the "data" folder. These files are the outputs from the ProgramAnalyzer script and therefore contain the loop features, as well as the times of the sequential and parallel versions that will be used to create the target. If the script is only being used to make predictions the timing information is not required and a script is made to handle that situation. However, if the intention is to create the models this data is required.

The script iterates through all the files and retrieves information from each loop of the benchmarks found in the files. For each loop, it retrieves its features, the loops ID, omp pragma, and problem size flag and calculates the corresponding target. The selected target was the speedup of the parallel version. To calculate this, the script averages the parallel and sequential versions' times and then divides the average sequential time by the average parallel time. If there are no valid time values found that loop is skipped.

The loaded data is organized into 4 main elements:

- loop_features - 2D list containing all the loop features. The 1st dimension separates the different loop and the 2nd separates each feature of that loop.
- loop_targets - list of all the loop targets.
- loop_info - 2d list containing all loops IDs, pragmas, and problem size flags. Its structure is similar to loop_features.
- feature_names - list of the name of every feature in the same order as they are read from the files,

The loop_features, loop_targets, loop_info, and loop info have the same order so that the same index in each is related to the same loop.

4.3.4 Data analysis

In this stage, the loaded data is analyzed to check the correlation of the features with themselves and the target. This process can be done before or after Data processing, by toggling the value of the parameters "data_analysis.pre_data_processing" and "data_analysis.post_data_processing" between true and false. There is also the option to visualize the correlation values as a heatmap.

To do this the parameter "data_analysis.data_correlation.graphical_results" must be enabled. The results are also stored in a CSV file in the "results" folder. The filename is dependant on when the data analysis is called:

- corr_matrix_pre.csv - the analysis is called before data processing.
- corr_matrix_classification.csv - the analysis is called after data processing for classification algorithms.
- corr_matrix_regression.csv - the analysis is called after data processing for regression algorithms.

This functionality will also be partly used in one of the feature selection algorithms in 4.3.5.

4.3.5 Data processing

Data processing is where the data is modified to be more readable to the algorithms and in turn, produce better models. The script supports 5 different operations that can be performed on the data: target processing, invalid input handling, data scaling, feature selection, and upsampling.

4.3.5.1 Target processing

This operation changes the values of the target. The changes depend on whether the processing is being done before classification or regression models.

In the first scenario, the operation occurs automatically and cannot be disabled. In this situation, the target, which is a float representing speedup at the time of performing this operation, is converted to a binary value, 1 or 0 using a simple operation. If the value is less or equal than 1 it is replaced by 0, otherwise, it is replaced by 1. This is done to convert the speedups into two classes, 0 is assigned to loops which are not supposed to be parallelized and 1 is assigned to the ones that are.

In the second scenario, the operation is optional. In the case of regression algorithms, the user has the option to define a threshold for the target. What this does is replace every value that is less or equal than the threshold by zero. This can be useful when there are a lot of very small speedup values that can act as noise in the algorithms. Thresholding these values makes the rest of the values stand out more and in turn help the regression algorithms. This can be enabled with the parameter "data_processing.threshold_target.enabled" and the threshold is controlled with parameter "data_processing.threshold_target.threshold".

4.3.5.2 Invalid input handling

This operation is very simple. It scans the values of features of all loops. If any invalid value is found it is replaced by a 0.

This operation can be enabled with the parameter "data_processing.handle_invalid_inputs".

4.3.5.3 Data scaling

Data scaling can be applied to the loop features to make them align better with the type of data the algorithms expect. The script supports 4 different scalers, all from the scikit-learn module. These can be selected with the parameter "data_processing.scale_data_algorithm". The following scikit-learn scalers are available, next to their corresponding parameter value: The table 4.6 presents the available scikit-learn scalers as well as their corresponding parameter value.

Scaler	Parameter value
Disable data scaling	-1
preprocessing.scale()	0
preprocessing.StandardScaler()	1
preprocessing.minmax_scale()	2
preprocessing.RobustScaler()	3

Table 4.6: Data Scalers

4.3.5.4 Feature selection

Feature selection is an optional operation that filters out unwanted features. This is usually done when there are a lot of features present but not all of them contribute positively to the model creation, possibly acting as noise or outliers. There are three different feature selection methods: variance threshold, target correlation, and prediction feature selection. They can be divided into two categories. The first two methods are used with data before it is fed to the algorithms to create the models and save the names and indices of the selected features. The last one is only used on data to be used to make predictions on backed up models. Feature selection can be enabled by using the parameter "data_processing.feature_selection_params.enabled". In both variance threshold and target correlation, a file is created with the parameters used for the respective method as well as the indices and names of the selected features. In the case of target correlation, the correlation of the selected features is also stored. This file is saved in the "results" folder.

A more detailed description of each method is presented below:

- Variance threshold - This method selects features based on their variance as the name implies. It is applied using the feature_selection.VarianceThreshold() from the scikit-learn module. It receives a value for the variance threshold and is then applied to given data. It removes every feature that has a variance inferior to the value provided. This method is used when the parameter "data_processing.feature_selection_params.algorithm" is 0. The parameter "data_processing.feature_selection_params.variance_threshold" controls the threshold value.

- Target correlation - This method relies on the correlation of the features with the target to select which features to keep. It can receive 4 parameters that are located in "data_processing.feature_selection_params.target_correlation":
  - use_threshold - if set to true, only features whose correlation with the target are above the specified threshold are selected. If set to false, the features are ordered by their correlation and only the first X are selected, X being specified in another parameter.
  - mode - mode can be "abs", "pos" and "neg" and its role depends on the value of the parameter use_threshold.

    If use_threshold is set to false, it changes how the top features are selected. If it is "abs", the features are ordered based on the absolute value of their correlation in descending order. If it is "pos", only positive values are allowed and they're ordered in descending order. If it is "neg", only negative values are allowed and they're ordered in ascending order.

    If use_threshold is set to true, it changes how the threshold is used. If it is "abs", the features whose absolute of the correlation value exceeds the threshold are select. If it is "pos", only the features whose values are both positive and larger than the threshold are selected. If it is "neg", only the features whose values are both negative and smaller than the symmetric of the threshold are selected.
  - target_correlation_threshold - float between 0 and 1 that acts as the threshold to select features.
  - top_x_correlation - positive integer that determines how many features are selected when use_threshold is false.
- Prediction feature selection - This method can only be used on data that will be used to make predictions. This is because instead of using an algorithm to determine which features to select, it reads the features to select from "model_processing_params.json" file of the selected model. This happens because to ensure the model works properly we need to feed it the same features like the ones used for the creation of the model.

4.3.5.5 Upsampling

Upsampling is a common operation applied when using imbalanced datasets. In our case, many more loops are not supposed to be parallelized than those which are. This can be a problem because an imbalanced dataset can lead to undesirable bias in models. Also, in these situations, it is common that the most important class is the under-represented one (which is our case). Loops that are supposed to be parallelized are under-represented but when we predict that a loop should be parallelized, we want to be confident in this prediction since a bad prediction can lead to results worse than not parallelizing at all. To help in this regard SMOTE upsampling is available.

In the case of classification, models SMOTE upsampling is mandatory and is built-in the pipeline. The SMOTE that is used for the classification is from the module imblearn. In the case

of regression models, SMOTE is optional and its use is determined by the parameter "data_processing.regression_smote". Since scikit-learn and imblearn did not provide SMOTE for regression algorithms the one used in this thesis was SMOGN [5].

In both cases, upsampling only occurs on the training data and never on the test data. This ensures that the metrics used to measure model performance only uses real data.

4.3.6 Data split

The data split for training and testing data occurs differently in classification and regression algorithms.

Classification algorithms use cross-validation so the data split is implicit in that part of the pipeline. We are only required to feed the algorithms the data to be used and they automatically split it.

Regression algorithms work differently. To be able to support the upsampling algorithm for regression, SMOGN, cross-validation had to be removed from the regression algorithms. This is because, as stated previously, upsampling can only be used in the training data. This wasn't a problem in the classification algorithms since the SMOTE algorithm used was supported by the Pipeline functionality from the imblearn module. This means that as the data split occurred inside the cross-validation, SMOTE could be applied only in the training data. Since SMOGN is not supported in this way, to use it the data needs to be split manually. The problem with functions that split data into a training and test set is that they rarely support stratification for regression models. Stratification is important because it ensures that the target values are equally or close to equally represented in the training and test sets. Using our case as an example, this prevents all low speedup entries from being in the training and all high speedup entries from being in the test set due to the randomness of the split. A situation like this can be highly detrimental to the models' performance. To solve this a data split function was developed that supports both randomness in the split as well as stratification.

The function works by ordering the loop feature and the targets by the ascending order of the target and dividing this list into smaller lists and finally applying data split into these smaller lists. Since the data is ordered and split occurs in much smaller samples of full data, we can have some confidence that the various target values will be more fairly represented in both data sets. This is accomplished since, by definition, for X entries that go to the training data, Y go to test data and these are always close in value since the split occurs inside the smaller lists. The function supports 6 parameters as listed in the table 4.7. This function was coded in a separate file[4] to allow other projects to use it without depending on the other scripts.

[4] https://github.com/joseloc300/AutoParSelector/blob/v1.1/python/utils.py

Parameter	Description
train_ratio	float between 0 and 1 (non-inclusive) that determines the ratio of a block that goes into the training set
block_size	int that represents the size of the blocks in which the data will be split after ordering
stratify	boolean that determines if the data split should be stratified or not
randomize	boolean that determines if the data inside each block is randomized before making the split
r_seed	int that is used as seed for the randomization
r_shuffles	int that determines the number of shuffles applied inside each block

Table 4.7: Regression data split parameters

4.3.7 Machine learning models

In this stage, the data is fed to various machine learning algorithms to create models. At the end of the project there were 5 models in active development and use:

- Regression:
  - SVR.
  - Ridge.
  - DecisionTreeRegressor.
- Classification:
  - C-Support Vector Classification - referred to in this thesis as SVC.
  - KNeighborsClassifier.

An important part of the development of these models is the tuning of their hyper-parameters. Hyper-parameters are values that affect the creation of the model and in turn their behavior and performance. These values should be adapted to the problem and data at hand. Tuning these parameters by hand can take a lot of time, so we automated the testing of these parameters. Each model has an associated JSON file in the "tuning" folder. Said JSON file contains a field named "param_grid" which is a list with a single dictionary inside. That dictionary contains the hyper-parameter values to be tested. The name of each field is the name of the respective hyper-parameter and it contains a list with all the values that should be tested for that hyper-parameter. After these values are loaded, the script tests the model with all possible combinations of hyper-parameters defined in the JSON file, saving the model with the best results, as well as the hyper-parameter values that achieved said results.

The regression models optimize for R^2 score and the classification models optimize for AUC.

R^2 score, also known as Coefficient of determination, is a value with no floor and a ceiling of 1.0, 1.0 being the best possible result. The value represents the proportion of the variance in the dependent variable that is predictable from the independent variable(s). This means that a value

of 1.0 means that the model explains all the variability in the dependant variable. A value of 0.5, for example, can be interpreted as 50% of the variance in the target variable can be explained by the features.

AUC is a performance metric for classification problems. ROC is a probability curve that plots two parameters: True Positive Rate and False Positive Rate. AUC measures the area under that curve and it tells how much a model is capable of distinguishing between classes. The higher the AUC, the better the model is at predicting them. The value ranges from 0 to 1.0 however in practical terms it ranges from 0.5 to 1.0. This is because if the model is scoring a value X below 0.5 we can simply invert the results and automatically we obtain a model with a score of 1 - X.

Despite those being the only measures used by the models to compare their performance when choosing which version to keep, the final results also show mean squared error for the regression models and several statistics like accuracy, precision, and recall for the classification models.

The creation of the models can be disabled in the parameters either by disabling a specific model or by disabling the category they belong to.

4.3.8 Predictions and Output

After the models are built, they can be used to make predictions on any loop if their respective features are provided. The predictions can be enabled with the parameter "make_prediction.enabled" and require the model's name, path, and a flag that tells if it is a classification or a regression model. It is important to point out that for the model to work, the data needs to have the same format as it had during the model's creation so it is advised to use the same parameters for data processing.

The final output is a JSON file with all the information related to the models' predictions as well as the information related to the loops that are being predicted. This is so the ModelTester script can relate each loop to its respective prediction and retrieve the pragma directly from this file without having to calculate them again.

4.4 Clava - ModelTester Script

In the following subsection, we detail how each section of the ModelTester script operates, explaining their inputs, outputs, and process of obtaining the latter from the former. The figure 4.3 depicts a high level view of the process. It can be broken down into a few different stages:

1. Retrieve the model's information - Load the information of the machine learning model's predictions and results.

2. Benchmark/program files search - The given source folder is searched for source files that are then stored in an object. A different object is created for each benchmark if the source folder represents a benchmark set. If it is a single program, a single object is created to store all info.

3. Measure model's performance - Add pragmas to their respective loops based on the info provided by the model. Add timers, compile the code, and run it to make time measurements for the model's performance.

4. Measure AutoPar's performance - Add pragmas to their respective loops based on AutoPar's naive method. Add timers, compile the code, and run it to make time measurements for AutoPar's performance.

5. Extract statistics and repeat - Statistics of the performance values measured previously, as well as data comparing how many loops each version parallelized, are saved. If there are more benchmark instances or more problemSizeFlags for the benchmarks, the previous steps are repeated for the new benchmark or problemSizeFlag.

6. Output - Output the results of the various performance measurements along with the data used.

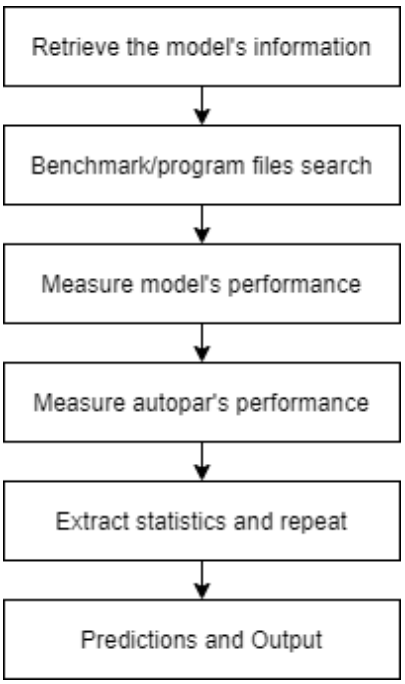

Figure 4.3: ModelTester stages

This script receives three inputs:

- originalParams - object containing the benchmark parameter info that is used in the current run of the script.
- resultsFile - path to file containing the results from the associated machine learning model.
- now - variable with the time when the script started.

The expected output is a file with all the information in modelTesterObj. This object stores most of the information used throughout the script as well as all results. The contents of this object will be later described in the Output section.

4.4.1 Retrieve the model's information

The first step when starting the script is creating a new modelTesterObj. The next step is to retrieve the information from the machine learning model that was chosen. This is done by loading the JSON file at the path specified by the script argument resultsFile. The information is loaded and subsequently copied to the modelTesterObj to equally named fields.

4.4.2 Benchmark/program files search

This stage behaves similarly to what was described in 4.2.1.

4.4.3 Measure model's performance

In this stage, the script retrieves all loops complying with the defined rules and adds their respective pragmas according to the predictions of the model being used. Finally, it adds the timers, compiles, and runs the code. The results are saved in modelTesterObj.

Firstly, it retrieves the for loops, but it uses a special function that is not used by the ProgramAnalyzer. This happens because the rules that determine what loops are allowed are different in this stage. In the previous stage, the goal was to gather info on as many loops as possible to have more information for the creation of the machine learning models. Furthermore, the pragmas were tested in isolation from other loop group pragmas. In this stage, the performance measurements are not of a loop group but the kernel of the program. The kernel is where most of the computation resides and therefore, is what we intend to improve. With this in mind, only the loops inside the kernel are suitable to be retrieved and parallelized. This zone can be defined in 2 different ways:

- #pragma kernel - if a #pragma kernel is found, only the loops inside it will be retrieved. It is expected that there is only one pragma with this name.
- function name prefix - if the previous option is not available, the script will search for loops inside functions whose name starts with "kernel". Although it can find a loop inside any function with this prefix it is expected that there is only a single function with this prefix. More details will be provided later.

Note that this search does not take into account loops that might be run due to function calls inside the pragma or the functions with the referred prefix.

After retrieving the loops, they are grouped into loop groups. The process is identical to the one in the ProgramAnalyzer script. The difference is that in this stage, we do not want to get all possible combinations of these loop groups, so each loop group is represented by a single list containing all suitable loops from that loop group. This is in contrast to the ProgramAnalyzer

script in which we would not only store every complete loop group but also every possible subset of that list.

The next step is to filter the retrieved loops based on the predictions from the model being used. The loops' id and problemSizeFlag are used to find their match in the prediction data. If both of them match, the script checks for the prediction the model gave for that particular loop. Since there are regression and classification models, the value is checked according to the type of model. If it is a classification model, the loop is removed if the prediction is 0. If it is a regression model, the loop is removed if the prediction is less or equal to 1.

After we have filtered the loops that the model predicted would perform badly, it is necessary to perform a second selection round. This time, instead of simply iterating loop by loop, it takes into account loop groups. This is to decide for each loop group what candidate to parallelize if there is more than one. The current iteration of this script supports only parallelizing one loop per loop group even though the ProgramAnalyzer script supported higher loop group sizes. If the model is a classification it chooses the outermost of the selected loops. If it is a regression it chooses the loop with the highest predicted speedup. The ids of the loops that were found suitable by the model and of those that were actually selected for parallelization are saved. This information is not only important to add the pragmas but also for section 4.4.6.

Afterward, the selected loops have their respective pragmas inserted. A timer is added to the source code to measure the time it takes to run the kernel of the program. There are two ways of adding them and they are similar to what was described earlier in this section. These are:

- #pragma kernel - if a #pragma kernel is found, the timers are added exactly before the pragma, and after the target of the pragma. The target of a pragma is the first statement after the pragma that is not another pragma, so if after the pragma there is a loop, the timer is inserted after the loop. As such, and as previously mentioned, the code expects only one pragma with this name.

- function name prefix - if the previous option is not available, the script will search for a function call whose name starts with "kernel". The timers are added exactly before and after the function call. This is why previously it was mentioned that only one function should have this prefix. Since it times the first function call with the prefix, having multiple functions with this name prefix, or several calls to the function, might lead to timing the incorrect zone of the code.

After compiling and running the processed source code, the timer results are extracted and are stored in the modelTesterObj.

4.4.4 Measure AutoPar's performance

In this stage, the script retrieves all loops complying with the defined rules and adds their respective pragmas using the naive method currently employed by AutoPar. Finally, it adds the timers, compiles, and runs the code. The results are saved in modelTesterObj.

Firstly, the loops are retrieved using the same function described in 4.4.3, since the rules for their selection are the same. This is necessary to ensure both approaches operate on the same loops, to make the results viable for comparison.

Following the loop retrieval, they are passed as an argument to the AutoPar function Parallelize.getForLoopsPragmas(), which attempts to parallelize and apply the appropriate pragmas to all loops passed to it as an argument. The return value is a dictionary separate into two groups, the loops that were parallelized and their respective pragmas, and the loops that weren't accompanied by an error message explaining this outcome. Although the pragmas are already applied, this still does not reflect the naive method currently used by AutoPar. This is because, at the moment AutoPar only parallelizes the outermost loop, when multiple nested loops can be parallelized. To apply this behavior, it is necessary to call the AutoPar function Parallelize.removeNestedPragmas(). As the name implies it removes the pragmas of nested loops and returns the ids of the loops whose pragmas were removed. This information will be relevant in the section 4.4.6.

The process of inserting the timers, running the code, and saving the results is similar to what was described earlier in section 4.4.3.

4.4.5 Extract stats and repeat

After acquiring the information of the loops that were and were not parallelized by each approach, they are used to calculate some statistics and save them in a benchStatsObj. These statistics include several values like the number of loops parallelized by one approach or the other or both. Some other values included are the number of loops parallelized by one approach but not the other and more.

This information can be useful in the analysis of the performance results by providing some insight into what might have improved or deteriorated performance.

When this is finished the benchStatsObj is added to the modelTesterObj in the field related to its benchmark and problemSizeFlag. Each of the values presented earlier is also added to a field in the modelTesterObj with a similar structure that tracks all the values described but global to all benchmarks and problemSizeFlags tested.

This determines the end of an iteration. If there are more benchmark or problemSizeFlags to test, the script repeats the steps from section 4.4.3 to this section. If all benchmarks have been tested already, the script moves to its final stage, the output state (section 4.4.6).

4.4.6 Output

This is the last stage of the ModelTester script. At this point, all benchmarks and problemSizeFlags are tested and the results are stored in the modelTesterObj. The contents of this object are dumped into a JSON file in the "model-performance" folder and the file is named after the model name and the parameter "benchGroupName". All the parallel versions tested have their binaries and source code saved in their respective subfolders inside the "outputs" folder.

4.5 Summary

We presented three separate tools that constitute the tool flow developed during this thesis. Two of them are coded in a mix of Javascript and LARA and make use of Clava while the other is coded in Python.

The first tool is the ProgramAnalyzer script and is responsible for analyzing source code and extracting the parallel loop, their features, and their speedup when parallelized.

The second is a Python script that loads the output of the ProgramAnalyzer and uses it to create machine learning models that predict if a loop should be parallelized or not. After the creation of the models, they are used to make predictions on the given loops.

The third tool is the ModelTester script. It loads the predictions and parallelizes the source code two times, one using the current AutoPar method and another based on the predictions provided by the models. In the end, the times are compared and output to a file for later analysis of the performance of both.

Chapter 5

Experimental Results

This chapter presents the experimental evaluation, what results were obtained, and the corresponding analysis. The models were evaluated according to their predicting performance and afterward, we analyzed how their predictions translate to actual speedup on the tested benchmarks.

5.1 Validation Methods

To evaluate the predicting performance of the models we used three benchmark sets: Polybench, NAS, and TEXAS_42[1]. Polybench uses all the problem sizes available, NAS uses the problem sizes class_S, class_W, and class_A. TEXAS_42 does not have multiple problem sizes. The ProgramAnalyzer script is used to extract the features and pragmas from the loops of these benchmark sets as well as calculate the speedup associated with each pragma. Only 1 pragma per loop group is used. Each algorithm creates 3 different models, each with a different dataset: one trained with only Polybench data, one trained with only NAS data, and finally one trained with both Polybench and NAS data. None of the created models uses TEXAS_42 data. When measuring the predicting performance of the final version of each model all the data retrieved from the ProgramAnalyzer script is used. This includes the data from the Polybench, NAS, and TEXAS_42 benchmark sets.

These are the number of loops used in each situation:

- Testing models:
  - Polybench data - 567 loops.
  - NAS data - 1245 loops.
  - Polybench + NAS data - 1812 loops.
  - Polybench + NAS + TEXAS data - 1842 loops.

[1]The source-code of the benchmarks used can be found here: https://github.com/joseloc300/AutoParSelector/tree/v1.1/clava/sources

- Comparing AutoPar heuristic's performance to the model's performance - 420 loops.

Polybench + NAS + TEXAS data represent all loop data extracted using the ProgramAnalyzer script.

The experimental setup used is comprised of:

- 2x Intel Xeon CPU E5-2630 v3 @ 2.40GHz

- 128GB RAM

To evaluate the performance of the proposed solution and validate its results, the AutoPar library runs several benchmarks using the developed solution for selecting which OpenMP pragmas to use, against the current naive method of always selecting the outermost pragma. The results without the solution will serve as the control. The objective is to check which and how many loops were filtered, and if this filtering improves the performance in comparison with the control results. The benchmarks used will be from PolyBench which is one of the industry-standard benchmarks sets for parallelization. Besides evaluating performance, these benchmarks will validate the results as they were made to simulate specific workloads representative of real-world scenarios. The fact that these benchmarks are widely used in the testing of parallelization tools also provides a way to compare the performance of AutoPar, with and without the solution, with other tools in future. The three versions of the models are helpful in this final measurement due to the way the data is used. During the creation of the models the training and test data are never mixed however when we get to the ModelTester script, all loops inside the kernel are considered valid to attempt to parallelize. This means that a loop that was used as training data for the model can be part of the final results. To provide a less biased view, three versions are tested. The first with only data from Polybench shows how a model educated on the loops fairs, the second with only NAS data shows how a model that has never seen any of the loops performs, and finally how it performs with the combined data from both.

The NAS benchmark set was intended to be evaluated as well but a recently discovered bug rendered the values unusable. The function that retrieves the loops for the ModelTester script did not have the intended behavior about the NAS benchmark. The problem is related to function calls inside the #pragma kernel. If there is a parallel loop inside a function being called inside another parallel loop, they should be considered as being part of the same loop group, which is something that is not currently happening.

The same happened in the TEXAS benchmark set which is why it is not featured in the final results as well.

This problem does not affect most of the Polybench benchmark set, since its loops, which are found based on the function with "kernel" prefix, do not have function calls. The exception is the benchmark "correlation" from the "datamining" folder which has also been removed.

All noninteger values are rounded to 5 decimal points.

5.2 Regression

In this section, we analyze the performance of regression models. The target variable of the regression models is the speedup of parallelizing a given loop. The metrics used to compare performance are:

- R^2 score - also known as Coefficient of determination, is a value with no floor and a ceiling of 1.00, 1.00 being the best possible result. The value represents the proportion of the variance in the dependent variable that is predictable from the independent variable(s). This means that a value of 1.00 means that the model explains all the variability in the dependant variable. A value of 0.50, for instance, can be interpreted as 50% of the variance in the target variable can be explained by the features.
- MSE - Mean squared error of the prediction of the target variable.

5.2.1 SVR

Looking at the performance with the test data (Table 5.1), SVR only performs well when using only the Polybench data, being the only version with a positive R^2 score. Even in this situation, the score is lower than what is desired at 0.42. This means that the features only explain around 42% of the variance in the speedup of the loops. Poor performance is also visible in the corresponding mean squared error value of 7.70.

Looking at the performance when predicting all the available data (Table 5.2), none of the versions are capable of doing so properly as none even has a positive R^2 score and the mean squared error values are in the millions. For the Polybench version, which was the best performing with test data, this means that model was not able to properly generalize as it fails when encountering new code bases.

Table 5.1: SVR performance with test data

Training set	R^2 Score	MSE
Polybench	0.42	7.70
NAS	0.00	100.38
Polybench + NAS	0.00	163.30
		End of table

Table 5.2: SVR performance when predicting all loops

Training set	R^2 Score	MSE
Polybench	0.00	5270031.21
NAS	0.00	5269962.73
Polybench + NAS	0.00	5269965.86
		End of table

5.2.2 Ridge

Looking at the performance with the test data (Table 5.3), like SVR, Ridge only performs well when using only the Polybench data. The score is slightly better with a value of 0.43 and the mean squared error is also slightly lower standing at 7.60. Even though the NAS and Polybench + NAS versions have poor performance and are not considered usable, it is important to note that out of the three regression models, SVR has the lowest mean squared error for these two versions.

Looking at the performance when predicting all the available data (Table 5.4), once again, none of the versions is capable of properly make speedup predictions, showing the same problems seen in SVR.

Table 5.3: Ridge performance with test data

Training set	R^2 Score	MSE
Polybench	0.43	7.60
NAS	0.00	21502.82
Polybench + NAS	0.00	16502.16
		End of table

Table 5.4: Ridge performance when predicting all loops

Training set	R^2 Score	MSE
Polybench	0.00	5270498.45
NAS	0.00	5272649.60
Polybench + NAS	-0.01	5296969.40
		End of table

5.2.3 DecisionTreeRegressor

In test data performance (Table 5.5), DecisionTreeRegressor does not fair as well as the other regression models. Like previously the best score is achieved when using Polybench data exclusively, with a score of 0.31 and a mean squared error of 8.67. Both these metrics are worse than the other 2 regression models. The NAS and Polybench + NAS versions have the best R^2 scores out of the 3 regression models, even though their mean squared error values are beaten by a big margin by SVR. Unfortunately, these values are still too low to consider the models usable.

Looking at the performance when predicting all the available data (Table 5.6), once again, none of the versions is capable of properly make speedup predictions, showing the same problems seen previously in SVR and Ridge.

Table 5.5: DecisionTreeRegressor performance with test data

Training set	R^2 Score	MSE
Polybench	0.31	8.67
NAS	0.00	37439.26
Polybench + NAS	0.00	25334.70
		End of table

Table 5.6: DecisionTreeRegressor performance when predicting all loops

Training set	R^2 Score	MSE
Polybench	0.00	5269032.01
NAS	0.00	5281664.63
Polybench + NAS	-0.01	5315721.01
		End of table

5.3 Classification

In this section, we analyze the performance of classification models. The target variable is divided into 2 classes, 0 and 1. 0 represents loops that should not be parallelized and 1 represents loops that should. The metrics used to compare performance are:

- AUC - Performance metric for classification problem. ROC is a probability curve that plots two parameters: True Positive Rate and False Positive Rate. AUC measures the area under that curve and it tells how much a model is capable of distinguishing between classes. The

higher the AUC, the better the model is at predicting them. The value ranges from 0.00 to 1.00 however in practical terms it ranges from 0.50 to 1.00. This is because if the model is scoring a value X below 0.50 we can simply invert the results and automatically we obtain a model with a score of 1.00 - X.

- Accuracy - Accuracy of the models' predictions.
- Precision 0 - Ratio of the predicted 0s that were true 0s.
- Precision 1 - Ratio of the predicted 1s that were true 1s.
- Recall 0 - Ratio of the true 0s that were correctly predicted as 0s.
- Recall 1 - Ratio of the true 1s that were correctly predicted as 1s.

5.3.1 SVC

Looking at the performance with the test data (Table 5.7), SVC performs the best, in terms of AUC, when using Polybench data exclusively. This is backed up by the rest of the results as it has the best values in all metrics. The NAS version has the second-best AUC followed by Polybench + NAS in last. Despite this, the Polybench + NAS version shows better accuracy and better Precision 1 and Recall 0 indicating it is better at predicting true positives and that it called more true negatives out of the negatives given.

Looking at the performance when predicting all the available data (Table 5.8), the NAS version becomes the best in terms of AUC, closely followed by the Polybench + NAS version with values of 0.65 and 0.63 respectively. The Polybench version which was the best with test data is now performing the worst with a very low AUC score of 0.53.

This shows that the Polybench version works well when used on a similar code sample but performs badly with different code bases. Even though it has the best accuracy, the results clearly show why its performance dropped so much. Recall 0 is nearly 100% and Recall 1 is little more than 5%. This means that nearly all true 0s were predicted as 0s and out of all true 1s only $\tilde{5}$% were predicted as 1s. In summary, this model is heavily biased towards predicting the value 0 which negatively impacts its performance.

By opposition, the NAS version performed the best in terms of AUC but has the worst accuracy of the 3 standing just below 50%. It has high Precision 0 and Recall 1 but low Precision 1 and Recall 0. This indicates that it is predicting too many 1s. This leads to high recall and low precision in that class and the inverse in the opposite class which is what we observe.

The Polybench + NAS is very close to the NAS version in terms of AUC coming in 2nd but has better results in the rest of the metrics in general. The accuracy is substantially sitting in at 0.65 vs the NAS version's 0.50. Comparing the 2, precision 0 is slightly lower but still very high at 0.87. Precision 1 is still low but slightly higher than the NAS version's value. The big difference between this version and the NAS is visible in the recall values. Contrary to the polarized values of the NAS version, the Polybench + NAS version shows values that are a lot more balanced, both

sitting at around $\tilde{6}4\%$. For these reasons, Polybench + NAS might be the better model of the 3 even though it does not have the highest AUC value.

Table 5.7: SVC performance with test data

Training set	AUC	Accuracy	Precision 0	Precision 1	Recall 0	Recall 1
Polybench	0.90	0.83	0.98	0.55	0.80	0.92
NAS	0.77	0.57	0.93	0.33	0.48	0.88
Polybench + NAS	0.64	0.63	0.86	0.33	0.63	0.64
						End of table

Table 5.8: SVC performance when predicting all loops

Training set	AUC	Accuracy	Precision 0	Precision 1	Recall 0	Recall 1
Polybench	0.53	0.79	0.79	0.55	0.99	0.06
NAS	0.65	0.50	0.89	0.28	0.41	0.82
Polybench + NAS	0.63	0.65	0.87	0.33	0.65	0.64
						End of table

5.3.2 KNeighborsClassifier

Looking at the performance with the test data (Table 5.9), SVC performs the best, in terms of AUC, when using Polybench data exclusively. As seen previously the Polybench version is the best in every metric measured with the exception of Recall 0. The NAS version is the worst in all metrics but follows the other versions closely with the exception of Recall 1. The Polybench + NAS version results sit between the Polybench and the NAS versions' results. The exception is the Recall 0 in which it performs the best of the 3.

Looking at the performance when predicting all the available data (Table 5.10), the Polybench + NAS version becomes the best in terms of AUC with a score of 0.73. The other 2 versions perform much worse in comparison with the Polybench version scoring 0.53 and the NAS version scoring 0.60.

Once more the Polybench version's AUC drops drastically from the highest value with test data at 0.93 to the lowest with all data with a very low score of 0.53. The reason for this drop is apparent when analyzing the other metrics, namely the Recall 0 and Recall 1. As seen previously in some models before, Recall 0 is nearly 100%, and Recall 1 is just over 6%. The model shows a

bias towards predicting the value 0 which negatively impacts its performance. A curious detail is that this version has the best Precision 1, which indicates that when a 1 is predicted we likely have higher confidence in that prediction compared to the other versions. The problem is the reduced number of times it actually predicts 1.

The NAS version is slightly ahead in second with an AUC of 0.60, which shows an improvement but not by much. It has the worst accuracy, Precision 1, and Recall 0 of the 3 versions. Despite this, it beats the Polybench version in AUC due to being less biased towards predicting 0. This is reflected in a much higher Recall 1 value of 0.33.

The Polybench + NAS is by far the best of the 3 versions. Not only does it have, by a considerable margin, the highest AUC with a value of 0.73, it also appears to have the most balanced model. It has the highest accuracy, precision 0, and recall 1 of the 3, the latter beating the rest by a large margin. The values in which it is not the best it follows the best very closely, namely in precision 1 and recall 0. These results indicate that this might be the best model out of all the models examined.

Table 5.9: KNeighborsClassifier performance with test data

Training set	**AUC**	**Accuracy**	**Precision 0**	**Precision 1**	**Recall 0**	**Recall 1**
Polybench	0.93	0.87	0.96	0.62	0.86	0.87
NAS	0.85	0.82	0.90	0.60	0.86	0.68
Polybench + NAS	0.88	0.84	0.92	0.62	0.87	0.74
End of table						

Table 5.10: KNeighborsClassifier performance when predicting all loops

Training set	**AUC**	**Accuracy**	**Precision 0**	**Precision 1**	**Recall 0**	**Recall 1**
Polybench	0.53	0.79	0.79	0.61	0.99	0.06
NAS	0.60	0.67	0.81	0.28	0.77	0.33
Polybench + NAS	0.73	0.81	0.87	0.56	0.89	0.52
End of table						

5.4 Comparison with AutoPar

From the previous results, it is clear that the classification models have a much better performance than the tested regression models. For this reason, the two classification models in their three versions are tested against the current AutoPar heuristic. Only loops inside the kernel functions are allowed to be parallelized and the time is measured from the start of the kernel function until its end. The performance metrics are divided into 5 different tables:

- Main metrics - displays speedup measurements and performance comparisons taking into account all the problem sizes tested. The speedups used are calculated with the AutoPar's speedup and the model's speedup. This means that the final value represents how faster the code is after being parallelized by the model compared to when parallelized by AutoPar. A value of 2.00 means that the code parallelized by the model is 2x faster than the code parallelized by AutoPar. This value should not be mistaken with the speedup calculated using parallel and sequential times. The average speedup is the arithmetic mean of the speedups. The table has 3 columns to show how each version of the model performs in comparison to each other. For the performance comparison metrics, it is considered a 5% margin of error, and the categories are divided in the following way:
  - Better performance - speedup > 1.05.
  - Comparable performance - 0.95 <= speedup >= 1.05.
  - Worse performance - Speedup < 0.95.
  - AutoPar DNF - AutoPar did not finish, but the model did. In this situation, speedup can't be calculated so we separate these instances from the other categories.
- Speedup by problem size (Polybench data) - Speedup data related only to the Polybench data version. Each column represents a different problem size and is supposed to provide a view of how each model fairs at different problem sizes.
- Speedup by problem size (NAS data) - Speedup data related only to the NAS data version. Each column represents a different problem size and is supposed to provide a view of how each model fairs at different problem sizes.
- Speedup by problem size (Polybench + NAS data) - Speedup data related only to the Polybench + NAS data version. Each column represents a different problem size and is supposed to provide a view of how each model fairs at different problem sizes.
- Parallelization statistics - Statistics related to what loops what parallelized. It displays results like the number of loops parallelized by the model, AutoPar, both, none, etc. Each column represents a different version of the model.

5.4.1 SVC

5.4.1.1 Main metrics

Starting with the main table 5.11, the Polybench versions appears to have the best results of the 3 versions. It has the best average and median speedup of the 3. The former being a really high value with a considerable margin from the other standing at 367.74. The latter is a very low value having nearly no difference from the others. The median speedup being very close to 1 might indicate that the performance improvements are not as good as the average suggests since it means half the values are under 1.00.

In the performance comparison metrics, the Polybench version has the best results once again, having the highest number of better performances and the lowest of worse performances. All versions have the same number of instances in which AutoPar did not finish. According to the results, all versions are better or at least comparable to the AutoPar method in more than 53% of the benchmarks run.

Table 5.11: SVC performance parallelizing Polybench vs AutoPar main metrics

Metrics	**Polybench**	**NAS**	**Polybench + NAS**
Average Speedup	367.74	118.79	132.30
Median Speedup	1.00	0.98	1.00
# Better performance	51.00	38.00	45.00
# Comparable performance	15.00	24.00	18.00
# Worse performance	38.00	42.00	41.00
# AutoPar DNF	12.00	12.00	12.00
% Better performance	43.97	32.76	38.79
% Comparable performance	12.93	20.69	15.52
% Worse performance	32.76	36.21	35.34
% AutoPar DNF	10.34	10.34	10.34
			End of table

5.4.1.2 Speedup by problem size - Polybench version

The table 5.12, shows how the Polybench version's speedup is affected when discriminating by problem size. Looking at the average speedups, although they are all above 1.00, we see a massive discrepancy across the various sizes going as high as 882.51 in the MINI problem size and as low as 3.28 in the LARGE problem size. This is not to say the 3.28 is a bad result, but only to demonstrate that the global average speedup that is presented in the table 5.11 is skewed due to extremely large values in the smaller problem sizes.

These results also show that this version works better with smaller problem sizes having 79.31% and 92% of speedups above 1.00 in the MINI and SMALL problem sizes respectively. In contrast, the STANDARD and LARGE problem sizes only have 28% and 36 of their respective speedups above 1.00.

Table 5.12: SVC performance parallelizing Polybench vs AutoPar by problem size (Polybench data)

Metrics	MINI	SMALL	STANDARD	LARGE
Average Speedup	882.51	499.50	3.30	3.28
# Valid Speedups Speedup	29.00	25.00	25.00	25.00
# Speedups above 1.00	23.00	23.00	7.00	9.00
% Speedups above 1.00	79.31	92.00	28.00	36.00
				End of table

5.4.1.3 Speedup by problem size - NAS version

The table 5.13, shows how the NAS version's speedup is affected when discriminating by problem size. Starting with the average speedups, this version behaves slightly differently from the Polybench version. The speedups can still be divided into 2 groups, the 1st comprised of the 2 smaller problem sizes and the 2nd with larger 2, and are lower compared to their Polybench version's counterparts. In this version, SMALL is the problem size with the highest speedup of 390.78 and MINI comes in second with 85.45. The STANDARD and LARGE speedup are far behind these 2 but relatively close to each other with values 2.14 and 2.12 respectively.

When looking at % of speedups that were above 1.00, this version is more interesting. MINI drops below 50%, while SMALL remains in 1st but with only 60%. The main difference appears in the 2 larger problem sizes. STANDARD improves slightly to 36% and LARGE has 52%. Focusing on the LARGE problem size specifically, this means that even though its average speedup is lower,

the amount of benchmarks that benefited from speedup is substantially larger, in this case, more than half.

Table 5.13: SVC performance parallelizing Polybench vs AutoPar by problem size (NAS data)

Metrics	MINI	SMALL	STANDARD	LARGE
Average Speedup	85.45	390.78	2.14	2.12
# Valid Speedups Speedup	29.00	25.00	25.00	25.00
# Speedups above 1.00	14.00	15.00	9.00	13.00
% Speedups above 1.00	48.28	60.00	36.00	52.00
				End of table

5.4.1.4 Speedup by problem size - Polybench + NAS version

The table 5.14, shows how the Polybench + NAS version's speedup is affected when discriminating by problem size. Starting once more with the average speedups, this version has the worst results. Except for the MINI problem size, all average speedup values are the worst of all 3 versions. All speedups averages are above 1.00 and the division between the 2 smaller and the 2 larger problem sizes is still visible. The STANDARD and LARGE values are especially low standing at 1.43 and 1.44 respectively.

When looking at % of speedups that were above 1.00, the 2 smaller problem sizes show an improvement when compared to the NAS version but still fall behind the values from the Polybench version. The STANDARD value is on par with the NAS version but the LARGE falls short of it, sitting in between the other 2 versions. In this version, only the 2 smaller problem sizes were able to achieve percentages above 50% with MINI and SMALL having 65.52% and 64.00% respectively.

Table 5.14: SVC performance parallelizing Polybench vs AutoPar by problem size (Polybench + NAS data)

Metrics	MINI	SMALL	STANDARD	LARGE
Average Speedup	153.60	369.32	1.43	1.44
# Valid Speedups Speedup	29.00	25.00	25.00	25.00
				Continued on next page

Table 5.14 – continued from previous page

Metrics	MINI	SMALL	STANDARD	LARGE
# Speedups above 1.00	19.00	16.00	9.00	11.00
% Speedups above 1.00	65.52	64.00	36.00	44.00
				End of table

5.4.1.5 Parallelization statistics

The table 5.15, shows some parallelization statistics of both the model's predictions and the AutoPar method. These values should help us better understand the values that were shown in previous tables. All versions are fed the same 420 candidate loops.

The Polybench version predicts that only 12 loops should be parallelized and of those only 4 are effectively parallelized. This is because of these 12 some are nested and therefore only the outermost is parallelized since we are working with only 1 pragma per loop group. By contrast, the AutoPar method parallelizes 204 loops. The model parallelizes 1 loop that AutoPar does not but in turn, does not parallelize 201 loops that AutoPar does. The 2 approaches have 3 parallel loops in common and of the 12 predicted, AutoPar parallelizes 6. The low number of loops parallelized explains the behavior of the model's versions earlier. The high average speedups in general but low in the larger problem sizes is because the model is barely parallelizing any loops at all. What happens is that in the smaller problem sizes, there is a large overhead when parallelizing. The Polybench version of the model does not parallelize those loops and since AutoPar parallelizes them, the high speedup values achieved come from avoiding the overhead of parallelizing the loops. This works in smaller problem sizes since most of these loops are best left sequential even if the parallelism is available due to the mentioned overhead the parallel version creates. However, when we get to the larger problem sizes this version's performance tanks because now, parallelism is more likely to be advantageous but this version is still barely parallelizing any loops. In this situation, what made the AutoPar method fail before, is what now gives it an edge. At these problem sizes, the naive method of parallelizing loops proves to be better than not parallelizing at all, which is nearly what this version is doing.

The NAS and NAS + Polybench versions show similar results predicting 230 and 206 loops respectively and parallelizing 103 and 98 respectively. AutoPar's results don't change since the method is the same. These 2 versions appear to be more liberal in what loops to parallelize which is a good thing since the opposite was the Polybench version's weak point. The number of loops predicted is similar to AutoPar's but the number of loops parallelized is a little below half of them. Of the loops parallelized by the NAS and NAS + Polybench versions, 93 and 92 respectively, were parallelized by both the models and AutoPar. Using this knowledge to re-analyze the information shown previously we can now understand why the NAS and Polybench + NAS versions were

much worse on the smaller problem sizes but considerably better in the larger problem sizes. These 2 versions parallelize many more loops than the Polybench version which in the smaller problem sizes is bad because, in those, most of the loops should be ignored to avoid overhead. However, when it comes to larger problem sizes, this amount of loop parallelization becomes a huge advantage, allowing the 2 versions to perform considerably better. Comparing these 2 versions, NAS performed better in the larger problem sizes whereas Polybench + NAS performed better in the smaller ones. Since the NAS version parallelized more loops, the same logic seems to apply. The extra loops decreased performance for NAS in the smaller problem sizes but provided extra speedup in the larger problem sizes.

Table 5.15: SVC parallelization statistics

Metrics	Polybench	NAS	Polybench + NAS
# Candidate loops	420	420	420
# Predicted by model	12	230	206
# Par by model	4	103	98
# Par by AutoPar	204	204	204
# Par by model but not AutoPar	1	10	6
# Par by AutoPar but not model	201	111	110
# Predicted but not par by model	8	127	108
# Par by AutoPar and model	3	93	92
# Predicted and par by AutoPar	6	107	114
			End of table

5.4.1.6 Overview

With all this knowledge, of the 3 versions, NAS seems to be the most balanced. It is the worst at the smaller loops but is still a decent filter for them and compensates with the performance on the bigger loops where it outperforms the other versions. NAS + Polybench can also be usable if the focus is more on filtering smaller loops, while also needing to select from some larger loops. The Polybench version is so reluctant to parallelize loops that it is difficult to use it in a realistic scenario. It is the best at filtering smaller loops which would give overhead but the results suggest

that this behavior is not so much indicative of a good performance filtering the smaller loops but more a bias that prevents it from parallelizing loops most of the time.

5.4.2 KNeighborsClassifier

5.4.2.1 Main metrics

Starting with the main table 5.16, all versions have a high average speedup value, the NAS version being slightly lower. This is in contrast with what we see in the SVC model in which Polybench had the highest average speedup while the other 2 versions trailed behind. Once again, the median speedups are all very close to 1.00 which means half the speedups were above this value and the other half were below. This is an indicator that possibly some benchmarks have high speedups that push the average up but aren't a frequent occurrence. The number of times AutoPar did not finish is 12, the same as seen with SVC.

Looking at the performance comparison metrics, the Polybench + NAS version has both the lowest percentage of worse performances and the highest of better being 27.59% and 45.69% respectively. In these metrics, the Polybench version follows closely in 2nd, and the NAS version is last. According to the results, all versions are better or at least comparable to the AutoPar method in more than 56% of the benchmarks run.

Table 5.16: KNeighborsClassifier performance parallelizing Polybench vs AutoPar main metrics

Metrics	Polybench	NAS	Polybench + NAS
Average Speedup	377.73	256.27	376.40
Median Speedup	1.00	0.98	1.00
# Better performance	49.00	45.00	53.00
# Comparable performance	19.00	20.00	19.00
# Worse performance	36.00	39.00	32.00
# AutoPar DNF	12.00	12.00	12.00
% Better performance	42.24	38.79	45.69
% Comparable performance	16.38	17.24	16.38
% Worse performance	31.03	33.62	27.59
% AutoPar DNF	10.34	10.34	10.34

End of table

5.4.2.2 Speedup by problem size - Polybench version

The table 5.17, shows how the Polybench version's speedup is affected when discriminating by problem size. Looking at the average speedups, the suspicions raised previously appear to be proven true. In the Polybench version the average speedup for the 2 smaller problem sizes, MINI and SMALL, are 922.63 and 494.63 respectively. Both in the hundreds with MINI being nearly 1000. By opposition, the 2 larger problem sizes, STANDARD and LARGE have very similar values close to 3, 3.25, and 3.22 respectively. Looking at the % of speedups above 1.00 it seems that this version isn't parallelizing enough as MINI and SMALL have high values, SMALL being 92%, but their performance drops with the larger problem sizes sitting near 30%.

Table 5.17: KNeighborsClassifier performance parallelizing Polybench vs AutoPar by problem size (Polybench data)

Metrics	MINI	SMALL	STANDARD	LARGE
Average Speedup	922.63	494.63	3.25	3.22
# Valid Speedups Speedup	29.00	25.00	25.00	25.00
# Speedups above 1.00	23.00	23.00	7.00	8.00
% Speedups above 1.00	79.31	92.00	28.00	32.00
				End of table

5.4.2.3 Speedup by problem size - NAS version

The table 5.18, shows how the NAS version's speedup is affected when discriminating by problem size. This version behaves similarly to the Polybench version but with overall worst values. All average speedups are lower and the same for the % of speedups above 1.00. The exceptions are STANDARD % of speedups above 1.00 which is raised from 28% to 36% and the LARGE % of speedups above 1.00 which is maintained at 32%. The loss in performance might be a consequence of this version parallelizing more loops as this seems to be the cause of loss in performance in the smaller problem sizes. This increase however is either not enough or isn't being applied in the correct loops since the performance in the larger problem sizes is still quite low.

Table 5.18: KNeighborsClassifier performance parallelizing Polybench vs AutoPar by problem size (NAS data)

Metrics	MINI	SMALL	STANDARD	LARGE
Average Speedup	544.08	431.13	1.90	1.91
# Valid Speedups Speedup	29.00	25.00	25.00	25.00
# Speedups above 1.00	20.00	18.00	9.00	8.00
% Speedups above 1.00	68.97	72.00	36.00	32.00
				End of table

5.4.2.4 Speedup by problem size - Polybench + NAS version

The table 5.19, shows how the Polybench + NAS version's speedup is affected when discriminating by problem size. This appears to be the best out of the 3 versions when analyzing speedup by problem size. For starters, it manages to obtain average speedups similar to the Polybench version which are its strong points. It also has similarly high MINI and SMALL % of speedups above 1.00 at 89.66% and 80% respectively, making it the highest value of this metric for the MINI problem size. With this said the main reason this version is considered the best in these metrics is that it does all this while also having the highest % of speedups above 1.00 for the larger problem sizes, both being 40%. Although this value is not very high, it is still worth noting that this improvement along the maintenance of the high speedups can indicate a better capability of predicting what loops to parallelize instead of simply being biased toward almost never parallelize any loop as we have seen before.

Table 5.19: KNeighborsClassifier performance parallelizing Polybench vs AutoPar by problem size (Polybench + NAS data)

Metrics	MINI	SMALL	STANDARD	LARGE
Average Speedup	912.81	500.21	3.44	3.32
# Valid Speedups Speedup	29.00	25.00	25.00	25.00
# Speedups above 1.00	26.00	20.00	10.00	10.00
% Speedups above 1.00	89.66	80.00	40.00	40.00
				End of table

5.4.2.5 Parallelization statistics

The table 5.20, shows some parallelization statistics of both the model's predictions and the AutoPar method. These values should help us better understand the values that were shown in previous tables. All versions are fed the same 420 candidate loops and AutoPar always parallelizes the same 204 loops.

The Polybench version predicts that 22 loops should be parallelized and parallelizes only 10 of those due to nesting. The behavior of this version is similar to the Polybench version of the SVC model. Very few loops are being parallelized which causes really high speedups in the smaller problem sizes but also lower speedups in the larger problem sizes. The speedup increase in the smaller problem sizes comes from avoiding the overhead that comes with parallelizing the smaller loops however, because of this bias, the model fails to properly parallelize in the larger problem sizes resulting in noticeably poorer performance.

Similarly to the SVC model's versions, the NAS and Polybench + NAS versions show similar results. The number of predicted loops is around half of those predicted by the same versions in the SVC model. The NAS and Polybench + NAS versions predict 100 and 93 loops to be parallelized respectively. Of these, around half are actually parallelized with the NAS version parallelizing 53 and the Polybench + NAS version parallelizing 40. These values confirm what was said before. The NAS version does parallelize more than the Polybench version, and as hypothesized, this is likely to lead to a drop in performance in the smaller problem sizes and an increase in the larger ones. The Polybench + NAS version parallelizes fewer loops than the NAS version as predicted before. This is likely the reason for its performance increase. It also furthers the hypothesis that it is better at selecting what loops to parallelize rather than simply relying on a bias. This is because the Polybench + NAS version predicts more than 4x the loops of the Polybench version and parallelizes 4x as many loops. Despite this increase in loops parallelized, we saw earlier that the average speedups were very similar to the ones from the Polybench version but the Polybench + NAS version had the benefit of substantially increased performance in the larger problem sizes. This means that the extra loops parallelized were able to maintain the average performance while also improving the speedups of several benchmarks in the larger problem sizes to values above 1.00.

Table 5.20: KNeighborsClassifier parallelization statistics

Metrics	Polybench	NAS	Polybench + NAS
# Candidate loops	420	420	420
# Predicted by model	22	100	93
# Par by model	10	53	40

Continued on next page

Table 5.20 – continued from previous page

Metrics	Polybench	NAS	Polybench + NAS
# Par by AutoPar	204	204	204
# Par by model but not AutoPar	3	10	10
# Par by AutoPar but not model	197	161	174
# Predicted but not par by model	12	47	51
# Par by AutoPar and model	7	43	30
# Predicted and par by AutoPar	10	47	35
			End of table

5.4.2.6 Overview

With all this knowledge, of the 3 versions, Polybench + NAS seems to be the best of the 3. It excels in most of the metrics and the results show that it is the best at the larger problem sizes in which models tend to struggle. Adding to this, the fact that it achieves this without sacrificing performance in the smaller problem sizes clearly distinguishes this version from the other 2 as superior. The NAS version was overall simply a worse version of the Polybench + NAS version and the Polybench version was too biased in not parallelizing loops. This bias appears to be good performance in the smaller problem sizes in which parallelization is usually bad but proves to simply be a bias due to the huge performance drop in the larger models and very low number of loops parallelized.

5.4.3 Best results

The best results achieved were by the NAS version from SVC and the Polybench + NAS version from KNeighborsClassifier. Both were chosen for appearing to be less biased and the best at predicting the loops to be parallelized despite the problem size used. When comparing the 2, the KNeighborsClassifier model has a considerably higher average speedup of 376.40 vs the SVC's 118.79. It also has a much higher % of benchmarks that showed better performance with 45.69% vs SVC's 32.76%. The same is seen in the average speedups by problem size in which the KNeighborsClassifier model beats the SVC model across the board. In terms of the % of speedups above 1.00 by problem size, the KNeighborsClassifier model is ahead in every problem size except for the LARGE problem size. Comparing the parallelization statistics, the main difference is the amount of predicted and parallelized loops with the SVC model being about double in

the two metrics compared to the KNeighborsClassifier model. This difference could explain why the KNeighborsClassifier model is superior in all but the LARGE problem size, where the extra parallelized loops likely make a positive difference in the speedup. In short, the KNeighborsClassifier model seems to be the best overall. However, if the focus is specifically on large loops, the SVC model has a smaller average speedup but was able to have a speedup above 1.00 in more benchmarks so it might be a good choice as well in this scenario.

5.5 Summary

For validation, the performance of the AutoPar library is compared with and without the developed solution. The benchmarks used are standard in the literature so that the results can also be compared with other similar tools in the future. The models are also analyzed in their performance with just the test data and with all data to judge how good they are at their predicting loops individually.

The regressions models are in general unusable. The best results achieved were only with the Polybench versions and only with test data. By contrast, the classification models performed fairly well with some achieving really good scores not only with the test data but also when using all data, showing they were able to somewhat generalize their learning and not become overfit.

For the comparison with AutoPar both classification models are used in their 3 versions. The Polybench versions of both models appear to be too biased in not parallelizing loops and are considered in general unusable. The remaining versions performed somewhat similarly, in each model. In the SVC model, the NAS version is deemed the best while in the KNeighborsClassifier model the Polybench + NAS version is deemed the best. The overall best model for general use is the Polybench + NAS version of the KNeighborsClassifier model.

Chapter 6

Conclusions

6.1 Conclusions

In this thesis, we set out to improve the heuristic used by Clava's AutoPar module to select and parallelize for loops. We proposed the use of machine learning algorithms to create models capable of doing this prediction.

To achieve this goal, 3 tools were developed to fulfill the tasks of creating, testing, and applying the models. These tools were made to support generic code bases as well as being heavily parameterized to increase the speed and ease of use.

The developed machine learning models had mixed results in terms of performance. The regression models performed poorly being nearly unusable, which might indicate the need for more data or the need for better features. The classification models performed very well on the test data and their performance dropped slightly when using all data.

The classification models were chosen to be tested against AutoPar. Several versions showed a big bias towards not parallelizing any loops which yielded great results in the smaller problem sizes but very poor performance in the bigger problem sizes. Two versions were able to somewhat overcome this bias and provide good filtering of the loops on the smaller problem sizes as well as decent loop selection for parallelization on the larger problem sizes.

The results obtained were far from perfect but show that this approach can be viable. Although regression has the capability of providing more information since it can predict speedup, the huge variation in the target variable in combination with an unbalanced data set made it hard to use these algorithms. Classification showed the best results and is likely the best approach in the near future. The results obtained were using only features retrieved from Clava and the source code AST. Adding data from more loops and extracting some more advanced features like the ones found in the analyzed literature are options with the potential to substantially increase these models' performance.

6.2 Future Work

Taking into account the field of study and the developed work, some proposals for future work are:

- Extend the developed tools to fully support more than 1 pragma per loop group. Currently, this is only partially supported by the ProgramAnalyzer script.
- Extend the ProgramAnalyzer script to extract more features, especially the ones that have already been found to be useful in the literature.
- Develop a mechanism that activates the pragmas in the code depending on the data size. This would be beneficial for loops that can either receive big or small amounts of data throughout the program's lifecycle. This would allow these loops to be parallelized in the former situation and prevent overhead in the latter.
- Test more programs and benchmarks, not only to retrieve data from more loops but to ensure diversity in the code bases analyzed. This can be an important factor to make sure the models can properly generalize the knowledge gained.
- Gather a more balanced data set. Currently, the data set is very unbalanced since most of the loops retrieved were not supposed to be parallelized. This can lead to a bias in the models like we saw in some of the versions tested. A more balanced data set can help mitigate this.
- Improve the customizability of the data processing stage in the Python scripts.
- Save more intermediary outputs for a better understanding of what is happening in each stage.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[2] Hamid Arabnejad, João Bispo, Jorge G. Barbosa, and João M.P. Cardoso. Autopar-clava: An automatic parallelization source-to-source tool for c code applications. In *Proceedings of the 9th Workshop and 7th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*, PARMA-DITAM '18, page 13–19, New York, NY, USA, 2018. Association for Computing Machinery.

[3] Hamid Arabnejad, João Bispo, João MP Cardoso, and Jorge G Barbosa. Source-to-source compilation targeting openmp-based automatic parallelization of c applications. *The Journal of Supercomputing*, pages 1–33, 2019.

[4] João Bispo and João M.P. Cardoso. Clava: C/c++ source-to-source compilation using lara. *SoftwareX*, 12:100565, 2020.

[5] Paula Branco, L. Torgo, and Rita P. Ribeiro. Smogn: a pre-processing approach for imbalanced regression. In *LIDTA@PKDD/ECML*, 2017.

[6] X. Chen and S. Long. Adaptive multi-versioning for openmp parallelization via machine learning. In *2009 15th International Conference on Parallel and Distributed Systems*, pages 907–912, Dec 2009.

[7] David Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.

[8] B. Liu, J. He, Y. Geng, L. Huang, and S. Li. Toward emotion-aware computing: A loop selection approach based on machine learning for speculative multithreading. *IEEE Access*, 5:3675–3686, 2017.

[9] X. Liu, R. Zhao, and L. Han. A compile-time cost model for automatic openmp decoupled software pipelining parallelization. In *2013 14th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, pages 253–260, July 2013.

[10] Aleksandr Maramzin, Christos Vasiladiotis, Roberto Castañeda Lozano, Murray Cole, and Björn Franke. "it looks like you're writing a parallel loop": A machine learning based parallelization assistant. In *Proceedings of the 6th ACM SIGPLAN International Workshop on AI-Inspired and Empirical Methods for Software Engineering on Parallel Computing Systems*, AI-SEPS 2019, page 1–10, New York, NY, USA, 2019. Association for Computing Machinery.

[11] Jeff Parkhurst, John Darringer, and Bill Grundmann. From single core to multi-core: preparing for a new exponential. In *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pages 67–72, 2006.

[12] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[13] S Prema, R Jehadeesan, and BK Panigrahi. Identifying pitfalls in automatic parallelization of nas parallel benchmarks. In *2017 National Conference on Parallel Computing Technologies (PARCOMPTECH)*, pages 1–6. IEEE, 2017.

[14] V. Purnell, P. H. Corr, and P. Milligan. A novel approach to loop parallelization. In *Proceedings 23rd Euromicro Conference New Frontiers of Information Technology - Short Contributions* -, pages 272–277, Sep. 1997.

[15] Paul Singleton. Performance modelling — what, why, when and how. *BT Technology Journal*, 20(3):133–143, Jul 2002.

[16] M. Tolubaeva, Y. Yan, and B. Chapman. Compile-time detection of false sharing via loop cost modeling. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, pages 557–566, May 2012.

[17] Zihang Zhang, Jingwei Sun, Jiepeng Zhang, Yuze Qin, and Guangzhong Sun. Constructing skeleton for parallel applications with machine learning methods. In *Proceedings of the 48th International Conference on Parallel Processing: Workshops*, ICPP 2019, New York, NY, USA, 2019. Association for Computing Machinery.