PERFORMANCE PREDICTION OF OPENMP PROGRAMS

A Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Victoria Stepanyan

May 2013

PERFORMANCE PREDICTION OF OPENMP PROGRAMS

Victoria Stepanyan

APPROVED:

Dr. Barbara Chapman, Chairman
Dept. of Computer Science

Dr. Edgar Gabriel
Dept. of Computer Science

D.Sc. Badri Roysam
Dept. of Electrical and Computer Engineering

Dean, College of Natural Sciences and Mathematics

Acknowledgements

I would like to thank my thesis advisor, Dr. Barbara Chapman, for giving me a rare opportunity to work in HPCTools research group and for her invaluable support in letting me proceed with the topic of my utmost research interest.

I cannot overestimate a contribution made by my mentor at HPCTools lab Sunita Chandrasekaran who guided my research efforts in a right direction, for inspiring me to discover new scientific prospects, and for her patience when reviewing my work and making valuable suggestions on further development of my research project.

I want also to express my gratitude to Dr. Edgar Gabriel, who showed considerable patience and willingness to help me with grasping HPC-related concepts.

Also, I would like to thank my HPCTools colleagues Sayan Ghosh, Deepak Eachempati, and others for their help and assistance.

Finally, I want to thank my dear family - my parents, my sister, and my husband for always being there for me and supporting me endlessly in all of my undertakings.

PERFORMANCE PREDICTION OF OPENMP PROGRAMS

An Abstract of a Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Victoria Stepanyan

May 2013

Abstract

OpenMP, a directive-based API supports multithreading programming on shared memory systems. Since OpenMP pragmas, directives, function calls, and environment variables are platform-independent, the API is highly portable. OpenMP provides necessary hints to the compiler in order to parallelize the given code, instead of focusing on the low-level details of the hardware.

Performance prediction methodologies enable estimation of performance factors (execution time, cache misses, effect of a compiler's optimizations) prior to the actual execution process. Existing approaches involve mathematical modeling of these performance factors. In order to achieve the best performance using OpenMP, it is critical to analyze cases such as the efficient cache utilization, optimal distribution of the workload among the CPUs.

We attempt to solve the problem of efficient per-thread workload distribution by predicting an optimal combination of an OpenMP scheduling policy and a chunk size (we call this combination a "class"). We employed PAPI hardware counters, R statistical package, machine learning software WEKA, TAU toolkit, and the OpenMP collector API. A set of heuristics were applied to analyze the data to find out the similarities between snippets of code pertaining to the same class. We developed a framework for taking measurements to gather the training data for the predictive model being constructed.

We evaluate our approach using several case studies from application domains such as Dense Linear Algebra, Structured, and Unstructured Grids. The results demonstrate that there is a set of parameters that influences the choice of the "class" for performance prediction.

Contents

1 Introduction **1**

1.1 Problem Statement . . . 2

1.2 Research Objectives . . . 3

1.3 Research Method . . . 4

1.4 Thesis Organization . . . 5

2 Existing Approaches to Performance Modeling **6**

2.1 Importance of Performance Prediction . . . 6

2.2 State of the Art . . . 8

3 Background to our Methodology **16**

3.1 OpenMP Programming Model . . . 16

3.2 Machine Learning Models . . . 21

3.2.1 Predictive Models . . . 21

3.2.2 WEKA Machine Learning Software . . . 23

3.3 Approaches to Predictive Performance Modeling . . . 25

3.4 Compilers . . . 26

3.5 Berkeley Dwarf Mining Programming Patterns . . . 28

3.6 Performance Application Programming Interface (PAPI) . . . 30

3.7 Performance Modeling and Analysis . . . 31

3.8 Gprof Profiling Tool 32

3.9 OpenMP Collector API 33

4 Performance Modeling 36

4.1 Overview of the Approach 36

4.2 Implementation Details 41

4.2.1 Target Loop Selection 41

4.2.2 Measurements Framework Introduction 48

4.2.3 Framework Implementation Details 50

5 Experiments and Findings 62

5.1 Testbed 62

5.2 Benchmarks Considered 63

5.3 Machine Learning Models Employed 65

5.3.1 Supervised Machine Learning Task Type 65

5.3.2 Supervised Machine Learning Algorithms 68

5.4 Code Instrumentation And Training Data Set Completion 72

5.4.1 Code Instrumentation 72

5.4.2 Training Data Set Completion 75

5.4.2.1 Execution Time and Hardware Counters Measurements Processing 75

5.4.2.2 Training Data Set Building and Runtime Configurations Categorization 81

5.5 Training Data Set Preprocessing and Experiment Analysis 88

5.5.1 Structured Grids Training Set 92

5.5.2 Unstructured Grids Training Set 100

5.5.3 Dense Linear Algebra Training Set 104

5.6 OpenMP CollectorAPI, TAU Profiling Toolkit 107

6 Conclusion and Future Work **110**

6.1 Conclusion 110

6.2 Future Work 112

Bibliography **114**

Appendices **116**

A Hardware Counters Available on Shark Platform **117**

B Framework for Taking Measurements **120**

List of Figures

3.1 OpenMP fork-join programming model 17

3.2 OpenMPdirectives format 18

3.3 WEKA Explorer tool screen-shot 24

3.4 Dense Linear Algebra Common Expression 28

3.5 Structured grids example 29

3.6 Unstructured grids example 29

3.7 TAU paraprof profiler 35

4.1 Diagram of our approach 40

5.1 DAG, BN example 69

5.2 Structured Grid dwarf, a model's accuracy vs. preprocessing techniques employed 100

5.3 Unstructured Grid dwarf, a model's accuracy vs. pre-processing techniques employed 103

5.4 Dense Linear Algebra dwarf, a model's accuracy vs. preprocessing techniques employed 107

List of Tables

5.1 Hardware counters measurements (execution time and class columns added), kmeans, DLA dwarf 80

5.2 Execution time measurements, chunk size equal to 2% of a problem size, myocyte, SG dwarf 84

5.3 Execution time measurements, chunk size equal to 5% of a problem size, myocyte, SG dwarf 85

5.4 Execution time measurements, chunk size equal to 10% of a problem size, myocyte, SG dwarf 86

5.5 Execution time measurements for a serial version, myocyte, SG dwarf 87

5.6 Per-thread exclusive execution time in a parallel region, % 108

Chapter 1

Introduction

An application's performance and parallelization are tightly interwoven. OpenMP API allows improvement of a program's execution efficiency. Rapidly evolving hardware makes performance analysis even more challenging. If an application showed considerable speedup when a certain OpenMP scheduling policy is used, this does not guarantee the same behavior on a different architecture. Moreover the speedup varies when the computation patterns vary. So it is essential to incorporate certain performance prediction techniques that relies heavily upon scheduling schemes.

Performance prediction can facilitate the process of software design space exploration process. More importantly, our approach not only allows choosing an appropriate scheduling policy but the chunk size used by an OpenMP runtime environment when assigning data to threads based on the scheduling applied. Not only is performance prediction able to make the process of developing an application faster, it can help to facilitate the process of making an application adaptable to the new architecture it might be tried on.

1.1 Problem Statement

Computer architectures currently available present a significantly diverse space of variations. Programming applications also evolve in their number and complexity. The range of problems which are solved using parallel programming paradigms widens from year to year. All the factors mentioned present a considerable amount of research which needs to be done before actually putting programming languages and OpenMP directives into use. This process can be extremely time-consuming. Therefore, it is of paramount importance to be able to model the effect of OpenMP directives usage beforehand. Specifically, OpenMP provides means of scheduling iterations of a loop for threads participating in the process. Execution time achieved highly depends on the combination of scheduling policy and the chunk size chosen. The combination which should be used to achieve desirable performance is affected by several factors - number of iterations (problem size), number of threads used, and compiler used to produce an executable, platform on which the application is going to be executed and execution time when no OpenMP scheduling was applied. We propose a method which would unite all those characteristics in a predictive model. The latter should take all the input information (the factors mentioned previously) and output a combination scheduling policy and chunk size, which would lead to the desired performance gain.

1.2 Research Objectives

The foremost intention of this research is to find a relationship between the factors mentioned in 1.1 and a combination of scheduling policy and a chunk size. We developed a predictive model which takes several parameters on the input and outputs a suggestion on which policy and a chunk size to use. Nowadays the spectrum of programming applications is immensely broad. This fact leads to different patterns of memory access, cache utilization and variation in computational complexity levels overall. Taking this into consideration, we made a decision to model performance in the scope of groups of applications called "Dwarfs" from Berkeley [1]. In particular, we worked with three of those patterns - Dense Linear Algebra, Structure Grids, and Unstructured Grids.

Our second goal was to classify the applications we tested so far into classes, each of which corresponds to a combination of scheduling policy and a chunk size. This classification allows us to locate a new application into one of the categories, thus providing a prediction on which policy we should use.

In our research, the following concepts present the factors mentioned in 1.1:

1. Hardware counters, to present hardware platform participating in the execution process;
2. Several compilers, to reflect the difference between the way in which compilers produce executable files and its influence on performance;
3. Problem size, to analyze how the number of iterations affects performance and

OpenMP directives choice;

4. Number of threads, to indicate an influence of system capabilities on the choice of OpenMP directives.

1.3 Research Method

In this section, we will give a big picture of our approach. The main goal of our method is building performance prediction model. This model must take an input and analyze it in such a way that the case being explored is located in one of the classes.

The input is presented by an application's signature, namely performance counters. The other part of the model's incoming data is characteristics of a specific case for which we want to get a prediction - the compiler used to build an executable file, number of threads which is going to be used, problem size, and execution time of a program when no scheduling policy was applied.

Having all this information in our model, it must produce an output - a classification of a given particular case in one of the classes. Each class presents the combination of scheduling policy and a chunk size.

To build a model we gather training data set - a number of various cases. For each of the cases we extract predictors values, the factors mentioned earlier. Also, we gather data when applying different combinations of policies and chunk sizes. After all the information is gathered for each case, we extract the best combination of scheduling policy and a chunk size. We do it by taking a combination which showed the shortest execution time. Next, a particular case is categorized into the class related to the combination.

After a training data set is ready, we train and validate our model using k-fold cross-validation method.

1.4 Thesis Organization

The rest of this document is organized as follows. In Chapter 2, we are discussing currently existing approaches to performance prediction and modeling. In Chapter 3, we introduce important background information related to OpenMP, machine learning predictive models, hardware counters, compilers, profiling tools, and an overview of programming patterns, "Dwarfs". Chapter 4 is dedicated to implementation details and an overview of techniques and methodologies used. In Chapter 5, we evaluate the resulting model by testing it using a k-fold cross validation strategy; we also present our measurements and explanation of our findings. Finally, Chapter 6 concludes the thesis and proposes ways to improve our approach.

Chapter 2

Existing Approaches to Performance Modeling

In this chapter, we are going to give a brief overview and summary of work from other researchers which is devoted to predicting the performance of parallel applications.

We will present an overall methodology existing at the moment, give a summary of works which we found the most relevant; and propose factors which made us go another way.

2.1 Importance of Performance Prediction

Several factors account for significance of performance modeling:

1. According to Moore's law, computer architectures develop at a very hectic pace providing software developers with new problems of proper utilization of the resources;

2. According to Wirth's law, software development is much slower than that of hardware. This leads to a mismatch between machinery resources and applications created. The mismatch, in turn, results in a complex software design space exploration process;

3. As new programing paradigms arrive, novel compiler software emerges proving a wide range of setting options for a program optimization.

All the reasons stated above make it very complicated to come up with an application which brings a desirable performance on the first try. A long process of application and platform system analysis together with an unavoidable process of trial-and error wastes a great amount of time and energy. In this sense, an ability to model or predict program's performance before actually running it comes as a remedy.

Existing approaches aim at an opportunity to find similarities between applications, platforms and runtime factors in order to classify a new program in one of the classes. Generally speaking, each class is a set of changes or settings which lets the application pertaining to a class perform at the best level.

Among the methodologies explored, the most popular are:

1. *Platform - oriented.* When a new architecture arrives, it is of utmost importance to be able to tune an application in such a way that it shows desirable performance. In this case, a prediction model is built for each application which is going to be run on this platform;

2. *Compiler - oriented.* To figure out the best combination of compilation and optimization flags of a given compiler for a given application, a prediction model is constructed;

3. *Derivative predictions.* This class of methodologies calculates execution time for a program with given features (both application features and platform characteristics); and after that, performance factors are extrapolated.

The next section presents an overview of research work which uses methodologies described earlier.

2.2 State of the Art

This work describes an approach to performance prediction in an architecture - independent manner [2]. The set of program characteristics is divided into two groups - application specific features and architectural characteristics. Static and dynamic types of analysis are used to predict an application's execution time. Static analysis gathers information on a Control Flow Graph (CFG) for every routine in a program, identifies loop nests, and schedule dependencies between instructions (both register and memory). Dynamic analysis is performed on an application's binary during runtime. Frequency histograms (depicts how often a particulate CFG edge is traversed), and memory access behavior monitoring information (measurements of memory reuse distances) are collected to describe the program from an execution standpoint. To construct a model of a program's behavior, data from multiple runs with different input parameters is collected. After dynamic data for an application is gathered, a CFG is constructed using information gathered at the stage of static examination. CFG is then used to determine paths in the graph and compute their frequencies. Execution cost in terms of instructions to be executed is computed for each of those patches. For each specific target architecture, an instruction execution time is calculated. Thus, by having the number of instructions and the execution time of each of them, the research team is able to predict the execution time of the program.

The final goal of the work presented in a paper is to model performance of a hybrid application, which is to be executed on both shared memory system and systems supporting MPI paradigm [3]. More importantly, they try to determine the most efficient combination of threads (OpenMP) and processes (MPI). The research team utilize both hardware profile (capabilities and resources of a given platform, network features) and the application signature (a set of operations performed by the program). The program in their approach is presented through its memory access pattern and carried mathematical operations. To predict an application's performance without actually running it, they first estimate communication time of a hybrid program, and then communication time of a pure MPI program is estimated. Message-passing efficiency is measured based on the ratio of the two communication times, the ratio is highly dependent on the number of OpenMP threads and MPI processes used. In the same manner, efficiency for an OpenMP program is used - a ratio of estimated execution time of serial and parallel versions is taken.

The work is devoted to a problem of hybrid MPI-OpenMP programs performance prediction based on memory bandwidth contention time and communication model, [4]. Their approach is focused on using memory bandwidth contention time on systems with a small number of cores in order to predict the performance on a larger architecture. First, they model performance of OpenMP applications. The modeling process makes the following assumptions:

- Memory bandwidth is considered to be the primary source of performance difference between a single core and a multi-core system;
- The parallel application's computation time can be divided into two types - time spent on computing using shared resources and time stalled on non-shared resources;
- When an application loads all the cores equally, time spent by a single core to compute results (Tc) does not change, time spent on memory bandwidth (Tm) increases when memory bandwidth per core is getting inefficient.

The first step the research team takes is to calculate execution times when one and two cores are used. When computing the latter a bandwidth ratio between baseline (one core) and two cores is used. The calculation includes both time characteristics described in the second assumption above. After that, Tc and Tm are found. Thus execution time of any number of cores can be calculated using Tc, Tm (which are found using execution times on one and two cores) and the memory bandwidth ratio between baseline (one core) and the target number of cores.

For modeling MPI program's performance, different combinations of number of OpenMP threads and MPI processes must be considered. The research team first defines an overlapping factor, which considers total execution time, time spent on computation only and communication time.

When both models for MPI and OpenMP are constructed, a performance of a hybrid system is computed as a sum of predicted OpenMP and MPI execution times multiplied by the overlapping factor.

Although the described approach requires few measurements, special care needs to be taken when configuring a hybrid system - one MPI process with OpenMP threads is run on each node.

The paper is devoted to a problem of computer architecture and optimizing compiler co-design [5]. The main goal is to predict the performance an optimizing compiler can achieve on any micro-architectural configuration. As a result, using the model, a new architecture can be tested for an optimizing compiler in order to pick hardware which will provide desirable timing. The research team choose a compiler (GNU one). They also pick 200 micro architectural configurations and 1000 compiler optimizations via random sampling. The runtime factors they are interested in - execution time, energy and the energy delay (ED) product. A model is created for each program for which the performance should be predicted. First, they run an application compiled on *-O1* optimization level on randomly chosen architectures. After that, values for performance counters are gathered. With the use of a machine learning algorithm known as Support Vector Machines (SVM), they try to find similarities between different architectural configurations in terms of the performance an optimizing compiler can achieve on them. Thus, to predict performance which can be achieved by a given application compiled with given optimization flags on a given architecture, the program's features needed to be located in one of the groups of similar architectural configurations.

As we see, this approach uses performance counters; however, the model has to be built for each application tested.

The work develops a technique which allows one to predict performance of a program on a given architecture when certain transformations are applied to the application [6]. The transformations that are considered in the paper are various compiler optimizations - loop unrolling, loop normalization, common subexpression elimination, etc. For a given set of programs which are to compose the training set of the model not all the possible transformations are considered. Only those optimization flags which discriminate speedups of the programs to a considerable degree are taken. The training data set of the model consists of speedups gained after randomly selected (from the pool of transformations which discriminate the programs the most), transformations are applied. First, for a given program a set of transformations is chosen in such a way that they characterize the program more accurately. The transformations are picked based on the following logic: by monitoring speedups which are gained after transformations are applied the research team tried to find out how similar this application is to the programs which are in the training set of the model. The ANN (Artificial Neural Network) machine learning algorithm is used.

The paper proposes an approach to determining good optimizations for a program based on values of hardware counters [7]. The final goal of the work is to build a predictive model, which, when receiving an input in a form of a set of hardware counters, outputs optimizations which will appear to be the most effective. A training set of the model is constructed as follows: hardware counters are collected for programs in a training data set; five hundred optimization flags sequences are randomly sampled and applied to each of the programs which are to train the model; speedups for the programs are collected; the ratio between the speedup of a current sequence and -*Ofast* optimization is computed, if the ratio is smaller than one the sequence of transformations is removed from the training set. As the result, they have a space of optimization flags sequences divided by a hyper plane in a multidimensional counters space. To predict which optimization flags are suitable for a given program, a logistic regression method is used. It matches hardware counters of a program being tested to counters of programs in a training data set. In other words, the model tries to locate counters of the application into one of the classes divided by a hyper plane. When this matching is found, the flags which appeared to give the best speedup for the program in a training set are said to be the best option for the current program.

Nevertheless, these methods discussed do miss some points.

According to the works we explored, a model should be built every time a new application/compiler/architecture arrives. This factor leads to a lack of flexibility and generality.

As a rule, the models featured in the related works try to predict performance for the whole program. This fact also makes the model not capture the peculiarities of computations and resource management employed by the program. If there is a need to predict performance for a part of a program certain amount work needs to be done in order to present the snippet being tested as a separate program.

Finally, the models proposed used few features to locate a program in one of the classes. A scarce number of characteristics of an application leads to possible low accuracy rate in case when there are classes which are similar to each other in terms of the attributes values pointing to that class.

In our approach we suggest the following points which employ best practices of performance modeling as well as proposing a novel strategy:

1. We would want to use hardware counters as we need a program's signature from the hardware level;

2. Machine learning models are showing accurate results with high accuracy rates; in this respect, we are going to utilize it for the classification challenge;

3. Our approach suggests using a greater number of program's features to be able to discriminate programs in a training more accurately;

4. The fact that we are building a model for every computation pattern in a set of chosen ones (3.5) allows us to state that the prediction will be more robust;

5. We also try to correlate our prediction to a compiler, which is used to build an application. This is important, as different compilers distribute resources and manage thread workload in different way. Including this factor in our model we will be able to capture influence a compiler has on distribution the workload among the threads iterations.

In this chapter, we presented several research works aimed at developing performance modeling strategies. Further, we would like to leverage certain points discussed as well as propose our approach to performance prediction problem challenge.

Chapter 3

Background to our Methodology

In this chapter we provide a reader with background information on the concepts and software tools we used in our research. We present an overview of an OpenMP programming interface. Further we discuss various predictive models offered by machine learning scientific field as well as the software tools used for machine learning approach. Next we talk about types of modeling different situations. We introduce compilers that we are using during our experiments, present programming patterns called "Dwarfs". Finally, we wrap up by discussion of the importance of performance modeling and profiling tools we employ.

3.1 OpenMP Programming Model

OpenMP API employs a fork-join execution model and provides a broad range of techniques to perform user-driven parallelization. This interface is available for C, C++ and Fortran programming languages.

The API is often referred to as a user-driven parallelization methodology because a programmer with the use of special compiler directives inserted in source code is able to execute an application on a multi-core system. More specifically, a programmer does not assign work to threads. The user, however, has various options as to how threads should execute a structured block of code. In the following few paragraphs we would like to present basic features of OpenMP as a programming paradigm and an interface.

As indicated earlier, OpenMP presents a fork-join programming model ([8]):

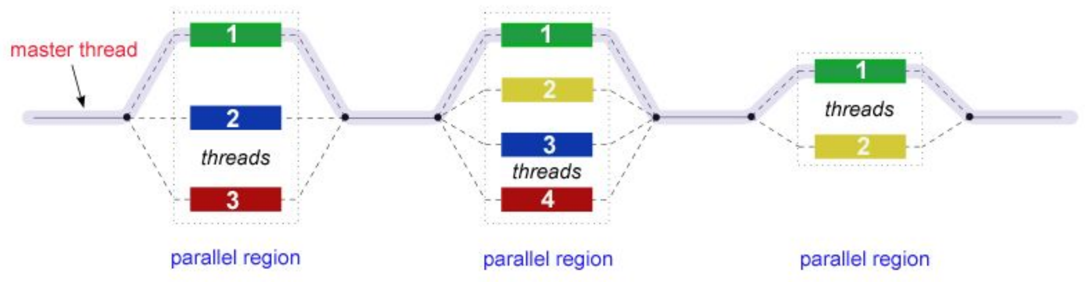
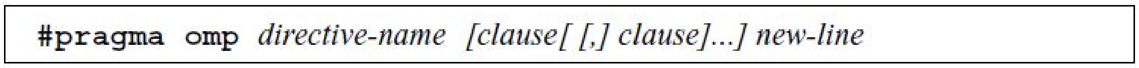


Figure 3.1: OpenMP fork-join programming model

Any program starts with a single thread named a master thread (3.1). Whenever a thread encounters *parallel* key word, a team of threads is created. After completing all the work inside a designated structured block of code, these threads are joined back into a single master thread.

OpenMP API employs a relaxed-consistency, shared-memory model. Each OpenMP thread has access to memory space shared among all threads in a current team, as well as its own private memory. At each point of execution a thread has a temporary view of variables which are visible to all other processors. However, its private memory space is not available to other participants.

Next we would like to give a brief overview of the most valuable OpenMP directives. The directives are being enabled by the *pragma* preprocessing key word. The structure of a directive is shown below:



```
#pragma omp directive-name  [clause[ [,] clause]...] new-line
```

Figure 3.2: OpenMPdirectives format

The most fundamental directives (3.2) in OpenMP API are *parallel* and *for* . As mentioned earlier, these directives enable forking a current process on a team of threads. In this work we mainly emphasize the importance of the latter directive mentioned. The *for* directive manages assigning iterations of a loop to threads for execution of the loop's body utilizing a multi-core system.

There are several *clauses* available for*for* construct in the API:

1. Private(list of variables). Each thread has its own copy of variables from the list and variables of other threads are not visible to a current one;

2. Shared(list of variables). Describes variables which are shared among all threads participating in the execution of a given block of code;

3. Lastprivate(list of variables), firstprivate(list of variables). These clauses indicate which variables are defined earlier in the code or by a master thread and that appropriate values should be copied to a thread's private memory; or if a

variable will be used after the joining took place and a value of a variable from private memory of a thread which executed the last iteration will be copied into that of master's thread;

4. Ordered. All iterations will be executed in a sequential order;

5. Schedule (kind [,chunk size]). Specifies the way in which a loop's iterations are divided among the threads.

In our work the last clause mentioned plays an exceptional role. *Kind* value specifies the way in which all iterations will be divided among threads; an optional *chunk size* value indicates the number of iterations that will be assigned to a participant, by default this is equal to 1. There are several scheduling kinds or policies in OpenMP:

1. Static. All iterations are divided into groups of chunk size and statically assigned to threads in round-robin fashion in the order of the thread number;

2. Dynamic. The chunk size is the number of iterations that are assigned to a thread whenever a thread is ready to request a new portion of work;

3. Guided. The same as dynamic scheduling with one exception - for a chunk size equal to 1 the size of a group of iterations assigned to a thread is proportional to the number of unassigned threads; for a chunk size greater than 1 the rule of forming groups of iterations is the same with one restriction - a group of iterations cannot contain fewer than chunk size iterations (except the last group of iterations which can contain fewer iterations);

4. Runtime. Is not a not a scheduling scheme per se. The scheduling policy and a chunk size are defined using special runtime variables.

The concept of scheduling iterations among the threads is of utmost importance since it leads to consideration of several aspects of parallel programming. Overhead of assigning work of chunk size iterations to a thread. Each kind of scheduling policies mentioned earlier has a separate mechanism of doing, moreover, different compilers and different platforms implement the very process of assigning differently. An end user might not be aware of all the peculiarities of a system one works on. Thus, picking a right policy and a corresponding chunk size have a direct relation to the speed of an application development process and an executable performance.

Furthermore, the number of iterations which are to be distributed and the number of threads that the application is going to be executed with are also points to be taken care of.

When solved improperly, a loop iterations scheduling problem ([9]) may lead to a considerable load imbalance, which is the main cause of resources wastage.

Our research is devoted to finding a way of optimizing a programmer's work. So far, the majority of OpenMP users have had to analyze all the factors on their own. The solution - a combination of a scheduling policy and a chunk size - may be far from an optimal one.

3.2 Machine Learning Models

In this section we want to present basic techniques of the Machine Learning approach which we found useful in the framework of our research.

3.2.1 Predictive Models

As stated in the Research Objective section (1.2) our foremost goal is to classify any particular case when applying a scheduling policy with a chunk size into a group of cases with similar features to find out the best combination of policy and chunk size. The problem of classification ([10]) presents a number of ways to recognize a particular case and place it into a group of similar ones. Any classification mold must *trained* , i. e. a sufficient amount of data should be gathered so that the model "learns" what are the classes presented and what is the distribution of provided data among the classes. After a model is trained, a new example which is not involved in a learning process is picked to be classified in one of the groups.

When dealing with classification problem we must explore two main directions - *supervised* and *unsupervised* learning methods. Supervised models allow one to take control of knowledge of a field of problems. Those models provide an advantage of tuning the model more accurately according to our knowledge of the classes, computation pattern involved, and the data we work with. Unsupervised learning models - cases or samples are grouped in classes based on similarities between among the samples. This type of modeling comes into practice when some input values are missing.

Further, we would like give a brief overview of machine learning algorithmic approaches.

Decision Tree-based methods. The ultimate goal of the approach is to divide the training data into groups of homogeneous members. This division is based on the most discriminative dividing criteria. The latter means either variance of a numeric output or the entropy of a group. The process of training the model stops when there is no significant gain in homogeneity. Finally, samples being a part of a group are the leaves of the tree. Those examples will vote for the prediction - the majority forms a class. A strong side of tree based methods is their flexibility. However, in terms of performance those methods are not among the best ones;

Linear regression-based methods. The main purpose of such an approach is to present an output (usually a numeric value) in the form of a linear equation where all the input variables have weights. Thus, the whole purpose of training such a model boils down to finding those coefficients. When input and output variables contain categorical values special care needs to be taken for presenting those factors in a numeric form. What is good about linear models is their high speed of learning, but on the other hand some input values require making an assumption which leads to inaccurate reductions;

Neural networks. A neural network can be considered a multiple layer of perception. Each layer is a logistic regression unit with multiple binary inputs and one binary output. Those models are able to learn non-linear relationships between an input and an output. As the output is presented in a binary form, categorical output variables need to be transformed into binary ones;

Bayesian network. This kind of algorithm is basically a dependency graph where each node represents a binary variable and each edge represents the dependency relationship. The learning process boils down to finding all joint probabilities of all incoming edges. Those models are highly scalable; however they require the data to be presented in a binary form;

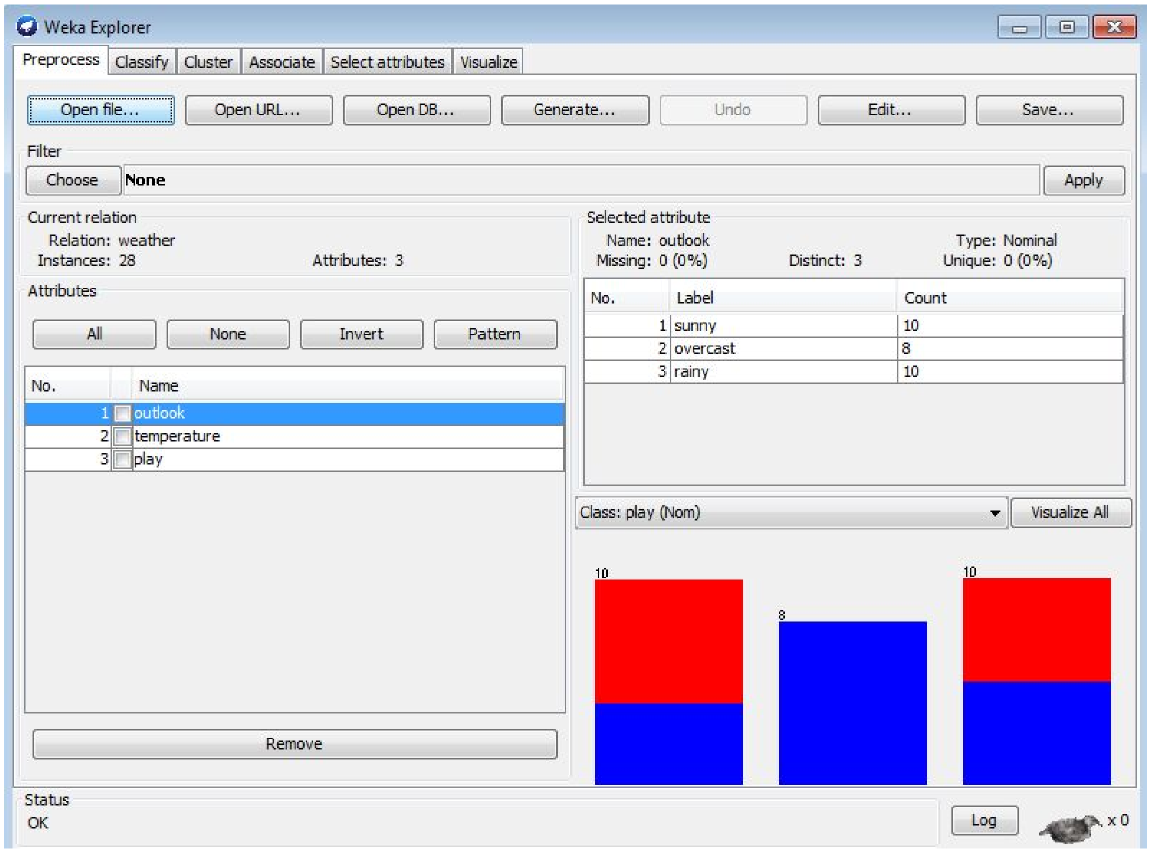
Support vector machine. The input for those models are numeric values, the output is presented in a binary form. At the heart of those algorithms is finding a linear plane with maximum margin to separate output classes. On a good side, those models are capable of dealing more than two classes;

Nearest neighbor. The idea of this method is to find a number of similar data points from the training set and the output is interpolated. As a rule, interpolation gives the majority values for categorical outputs and an average for numeric ones. As no model needs to be trained, those algorithms are considered to be simple in use. Yet, the data needs to be presented in a form of a distance-aware tree. Another weakness of this approach is the fact that it cannot handle large numbers of dimensions of the output.

3.2.2 WEKA Machine Learning Software

WEKA is a machine learning workbench which provides an environment for solving classification and regression problems. The most commonly used part of this software is an Explorer tool.

Explorer (3.3) is an interface which allows loading data and performing different



Filter: None

Current relation
Relation: weather
Instances: 28
Attributes: 3

Attributes

No.	Name
1	outlook
2	temperature
3	play

Selected attribute
Name: outlook
Missing: 0 (0%)
Distinct: 3
Type: Nominal
Unique: 0 (0%)

No.	Label	Count
1	sunny	10
2	overcast	8
3	rainy	10

Class: play (Nom)

Status
OK

Figure 3.3: WEKA Explorer tool screen-shot

kinds of analysis on it:

1. Visualize the data with respect to different attributes;

2. Solve classification problems - build a model base on a loaded data set and classify a new test set of data points. The output shows classification and errors of predictions. That can enable the user to find the most effective models for a particular problem;

3. When performing classification or regression various methods (comprising the ones discussed in 3.2.1) are available;

4. We can also test the model using cross-validation option;

5. Clustering algorithms for dividing an input dataset on clusters is available as well;

6. One of the most important features of the WEKA Explorer interface is selecting attributes. This techniques evaluates all the attributes and indicates which ones are the most important when predicting an output value.

In our research we use WEKA Explorer's interface to evaluate different machine learning models in order to pick one with the lowest error rate, and to evaluate our model on test cases when trying to locate them into one of the output classes.

3.3 Approaches to Predictive Performance Modeling

As mentioned earlier in this work, at the heart of our approach there is building a predictive model. It must take an input of several factors and output a combination of scheduling policy and chunk size. The model should output the prediction so that design space exploration process speeds up because there is no need to run an application trying out different combinations of policies and chunk sizes.

Further we would like to give a short summary of two fundamental approaches to modeling ([6]).

1. *Static Program Feature - based modeling.* This method assumes that one can

identify a set of static program features which can characterize a program's behavior. A good side of this approach is that to predict an application's behavior we don't have to execute it. However, it is very hard to extract right features which would statically reflect how an application would behave;

2. *Reaction - based modeling.* This kind of algorithm involves an empirical analysis of a program. One has to test or "probe" an application to observe its reaction and after that look at similarities between this reaction and reactions of previously executed programs. This approach allows us to perform a more objective analysis - we don't have to extract the features. We made a decision to use this method in our research.

3.4 Compilers

It is a well-known fact that OpenMP based parallel applications heavily depend on how efficiently the object code code is generated; how well the work is distributed among the threads. All this is managed by a compiler. In our work we use three compilers - OpenUH, PGI compiler and GNU compiler. Further, a brief overview of each of the compilers follows.

1. *OpenUH Scientific Compiler*, [11]. OpenUH is an open source portable OpenMP compiler suitable for C/C++/Fortran98 languages. OpenUH compiler is based on Open64 compiler. The compiler uses a different approach to translate from Open MP directives code into multi-threaded code with runtime library codes.

What is different in OpenUH is the fact that the code inside the parallel region is transformed into a micro task and then is encapsulated into the original parallel region. The advantage of this method is that the variables used inside the parallel region are still visible to all threads, thus making them shared. Another strong side of the compiler is it is an optimizing compiler. What's more, OpenUH attempts to optimize both original code inside the parallel region and the micro task generated;

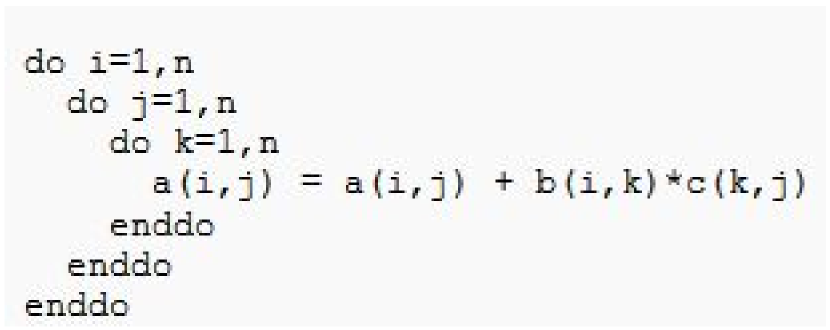
2. *GNU Compiler*, [12]. Is a part of GNU Compiler Collection (GCC). It has been ported to various architectures. Also it is adopted as the standard compiler for most Unix-like computer operating systems. It covers front ends for the following languages: C (gcc), C++ (g++), Objective-C, Objective-C++, Fortran (gfortran), Java (gcj), Ada (GNAT), and Go (gccgo). OpenMP implementation is one of the most significant parts of the compiler. This compiler cannot be omitted because it is the standard accepted by various vendors thus making it possible to port a model which included measurements from GNU compilers on a wide variety of programs;

3. *PGI Compiler*, [13]. Incorporate global optimization, software pipelining, and vectorization and shared-memory parallelization facilities. Supports OpenMP and MPI APIs. Has been accepted as a compiler producing fast code for high-performance applications by scientific and engineering communities;

3.5 Berkeley Dwarf Mining Programming Patterns

Patterns of computation are of crucial importance when dealing with performance modeling. The way in which a computation goes shapes memory access, certain resources usage. Thus, applications pertaining to different patterns have differences in their application signature, mostly in the level of hardware counters. A set of counters revealing for one pattern may not be significant of others at all. At this point we would like to introduce the "Dwarf" ([1]). The dwarfs refer to thirteen classes of computation and communication patterns. Members of each group are similar in computation and data movement. Considering the list of the patterns, we decided in favor of the following ones:

1. *Dense Linear Algebra (DLA).* Classic vector and matrix operations comprise the class. Usually, data is laid out in a form of arrays, operations are done on rows and columns. If one tries to map the applications pertaining to the class onto a multi-core system, data distribution issues come as the first priority in achieving performance gain. The figure 3.4 outlines a common set of operations which can be considered a Dense Linear Algebra pattern:



```
do i=1,n
  do j=1,n
    do k=1,n
      a(i,j) = a(i,j) + b(i,k)*c(k,j)
    enddo
  enddo
enddo
```

Figure 3.4: Dense Linear Algebra Common Expression

2. *Structured Grids (SG).* Data is arranged in regular grids (3.5). The sequence of grid update steps comprises the computation. At each step all points are updated using information from neighboring points. When mapping onto a multi-core system a contiguous sub grid can be assigned to a processor. In this case each thread has to extract only neighboring nodes on the grid.

$$u'[i,j] = \frac{1}{4}(u[i-1,j] + u[i+1,j] + u[i,j-1] + u[i,j+1]) - h^2 f[i,j]$$

Figure 3.5: Structured grids example

3. *Unstructured Grids (UG).* The computation is performed on irregular grids (3.6). The dataset is usually presented as a mesh covering surfaces of objects. The mesh can be transformed to a graph whose edges represent the geometric nearest-neighbor relationship between mesh points. When mapped onto a multi-core system the irregular grid can be divided into sub-meshes. The only communication on the sub-groups level is that of neighboring nodes. With the use of graph partitioner each processor is assigned a portion of a mesh in such a way that as few edges are crossed between processors as possible.

$$\mathrm{A}'[\mathrm{B}[\mathrm{C}[\mathrm{i}]]] = f(A[B[C[i+1]]] + A[B[C[i+2]]] + A[B[C[i+3]]])$$

Figure 3.6: Unstructured grids example

The factors which caused us to pick the patterns indicated earlier are:

- We look for the patterns which are most commonly used by programmers in various domains of problems;

- Easily distinguishable patterns in code;

- All three patterns selected have a significant degree of parallelization.

3.6 Performance Application Programming Interface (PAPI)

The PAPI interface provides developers with a tool for getting use of performance counters ([14]). The counters reside in a small set of registers named *Events*. Each event represents a special signal related to a processor's function. Taking values of those counters into account allows developers to see how the source code is mapped onto a target architecture. There are two types of the events: native and predefined. Native events are the signals which are measured by the CPU, those events can be accessed only through the low-level interface. Predefined or preset events are about one hundred events which are derived from the native ones. This correlation facilitates the process of performance analysis including performance tuning, compiler optimization, debugging, monitoring and performance modeling.

PAPI consists of two interfaces for monitoring the counters: low level and higher level interfaces.

The high-level interface allows to start, stop and read a specific event at a given time for a single-threaded application. It enables taking measurements of present events only which limits its use when derived events need to be investigated.

The low-level interface allows a programmer to handle the counters in groups called *EventSet*. The low-level interface also allows to take all necessary measurements for each thread in a multi-threaded application. In addition,the low-level interface is far more programmable than the higher one, that allows making measurements finer grained. Finally, the overhead incurred is considerably low.

Having considered the factors above, we opted for the low level interface.

Several steps need to be taken to make use of the low-level interface:

1. Initialization of the low-level interface - sets up PAPI library;
2. Create EventSets - create an integer handler for the set of events;
3. Add events to the created EventSet;
4. Start, Read, Stop Events in a set : start - starts counting the values of events in a specified EventSet; read - copies the values of events in a specified EventSet into an array; stop - stops the process of counting the values of events in a specified EventSet and returns the measured values.

3.7 Performance Modeling and Analysis

Performance is a major factor to consider when assessing the result of a working application. Performance modeling in turn provides a way to tune an application on an early design stage so that the outcome is closer to the desirable one. When approaching performance modeling one has to identify the application scenario. If a model is being created for a class of programs - the similarities between members of the class should be captured. This is the point when hardware counters may come into use - logically, each class of programs must have a set of counters responsible for reflecting a pattern of computation.

Performance modeling is of utmost importance because it is one of the major stages of software design space exploration. A great amount of time, human and financial resources are saved when future performance can be estimated. Here we would like to present certain benefits provided by performance modeling:

- Eliminate trial-and-error approach to tuning application's performance;
- Design process is more purposeful when performance considerations are made in the early stages of design;
- Performance modeling enables developers and scientists to estimate shortcoming of the code in order to avoid backtracking in future;

In our research we investigate a new way of performance modeling in order to get an insight into what factors play a role in picking a scheduling policy.

3.8 Gprof Profiling Tool

Since we are interested in a particular snippet of code, we have to profile the application in order to identify the loop that we are interested in. The details of loop identification problem are discussed in the next chapter.

Gprof is a profiling tool providing various forms of application analysis:

- *Flat profile.* This type of profiling outputs execution time spent on each function;
- *Call graph.* For each function it's callers and callee are displayed;
- *Annotated source.* Outputs the copy of source code; each line is annotated with the number of times it is executed;
- *Line-by-line profile.* All lines of the source code are sorted by the percentage of the total execution time spent on this line;

In our work we are going to use the last option since we would want to analyze the code from within.

3.9 OpenMP Collector API

On the final stage of our research we explored a few applications from different dwarfs using OpenMP Collector API. Since our work is tightly related to an OpenMP parallel programming paradigm, profiling of OpenMP code can provide a number of valuable insights.

The Collector API ([15]) implements communication with the OpenMP runtime and gather information about a program's execution details. In a nutshell, it registers events by using *__omp_collector_api routine* routine and allowing backtracking each event. Every time a program encounters an event (a particular state during the execution process), The OpenMP runtime library verifies whether the event is registered by the collector tool. The collector being used in our work is implemented the OpenUH compiler ([11]).

To visualize and analyze the information gained by the collector tool we use TAU profiling toolkit.

TAU (Tuning and Analysis Utilities) ([16]) enables researches to investigate performance information through code instrumentation. Those instrumentation can be inserted manually or automatically via Program Database Toolkit (PDT). OpenUH runtime has the collector API incorporated in in the compiler that is why when configured properly the TAU toolkit uses UH collector API instead of the default PDT.

The information that the collector provides includes implicit barriers for each parallel region; exclusive/inclusive execution time for each thread participating in a parallel region.

TAU toolkit also provides performance information analysis and visualization. Considering picture 3.7 we can observe the mean of execution time spent on various functions; we can monitor the proportion of time each thread spent on execution of a particular function. After compiling the program with TAU compilation script *tau_cc.sh* and execution of the generated program a number of application profiles are created. Those profiles are analyzed with *paraprof* profiler which encapsulates TAU methodology:

Using the collector API we are able to see all implicit barriers in all parallel

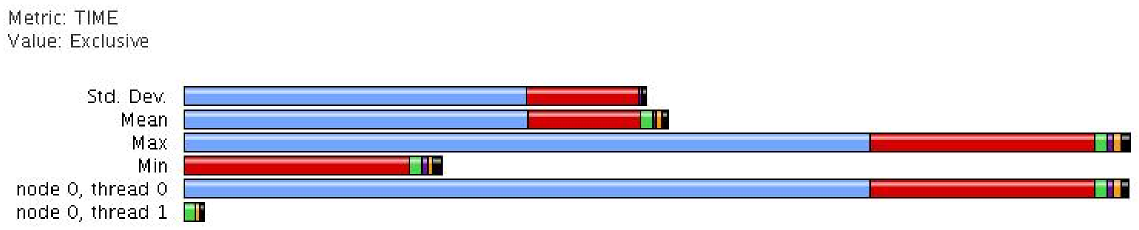


Figure 3.7: TAU paraprof profiler

regions of the program. That helps us to understand the behavior of the application better.

Chapter 4

Performance Modeling

In this chapter we are going to introduce and give a detailed explanation of our approach - taking measurements, analyzing received data, and building the predictive model.

4.1 Overview of the Approach

The purpose of the following section is to provide a reader with a brief introduction to the approach and to present all the steps we take to come up with the result.

Since OpenMP scheduling is applicable to ¨for¨loops, the very first thing that we would want to do is to identify the loop which pertains to a certain dwarf (4.2). Our final goal is to build a predictive model. The model should find out how similar different loops belonging to chosen dwarf are to each other. By similarity we mean how close their performance measurements are. Applications with similar features form a separate class - the same combination of scheduling policy and chunk size show suboptimal performance for those loops. To find similarities between the loops we have to characterize them by taking an application's signature (1.3). For that purpose we measure hardware performance counters available on the machine using a PAPI interface (3.6); we also measure execution time for the loop when no scheduling policy is applied. The latter characterizes the application's default behavior on the platform when no scheduling is going on.

Since each class stands for a particular combination of OpenMP scheduling policy and a chunk size we have to determine which combination appears to be suboptimal for a particular case of execution configuration. By execution configuration we mean the following factors:

- A compiler, which is used to produce an executable file;
- The number of threads used;
- Problem size.

Thus, we have to consider each of the configurations and find out which combination of scheduling policy and a chunk size produce desired performance. In order to do so we have to take measurements of execution time of the selected loop when various combinations of scheduling policy and a chunk size (including no scheduling case) are applied.

After the timings are gathered, for each of the cases we define the best combination of scheduling policy and chunk size, the one which leads to minimum execution time. This combination forms a class - if for an particular configuration this combination appears to be the most beneficial, it belongs to a corresponding class, we ¨label¨this configuration.

Machine learning predictive models require a data set which is used for training process - from this training data set an algorithm learns which values of the features point to which classes. In order to form the training set we gather information on hardware counters. Using this data set our model *learns* what are the classes (combinations of scheduling policies and chunk sizes) and what are their characteristics (execution time when no scheduling is applied, execution configuration and hardware counters). In other words, for each dwarf, our model divides the space of provided cases into classes and tries to match each case's characteristics with the class. More importantly, our model tries to find out similarities between cases when the same combination of scheduling policy and chunk size appear to be the most beneficial.

Our model is supposed to work as follows:

1. Take an input data: compiler, number of threads, problem size, execution time with no scheduling applied, and values of hardware counters. These features are supposed to characterize a particular execution configuration. OpenMP API is highly implementation-dependent, each compiler has its own heuristics to implement the API's details including scheduling policies. Number of threads affects the workload distribution because each thread gets a portion of work

to perform, thus the number of participating threads and their features (which thread is faster/slower) is important. Execution time with no scheduling applied characterizes default application's behavior. Hardware counters provide an insight into low-level details of execution of a particular loop when various scheduling techniques are applied;

2. Determine, to which class in a training data set the current input is closer;

3. Output the result - the class to which an application pertains.

A big picture of our approach can be represented by the diagram in Figure 4.1. First of all, we explore various computation patterns considered three of the dwarfs for our research (3.5). We consider applications pertaining to a certain dwarf in order to identify computationally intensive loop which employs a particular pattern. We instrument our code with the framework we developed in order to gather information (take performance measurements) which is going to be used by our model as a training data set. After the last step is completed a predictive model is built and evaluated for each of the selected dwarfs. Finally, a set of techniques are applied to improve the model's accuracy.

Further, a detailed explanation of implementation techniques and methodology follows.

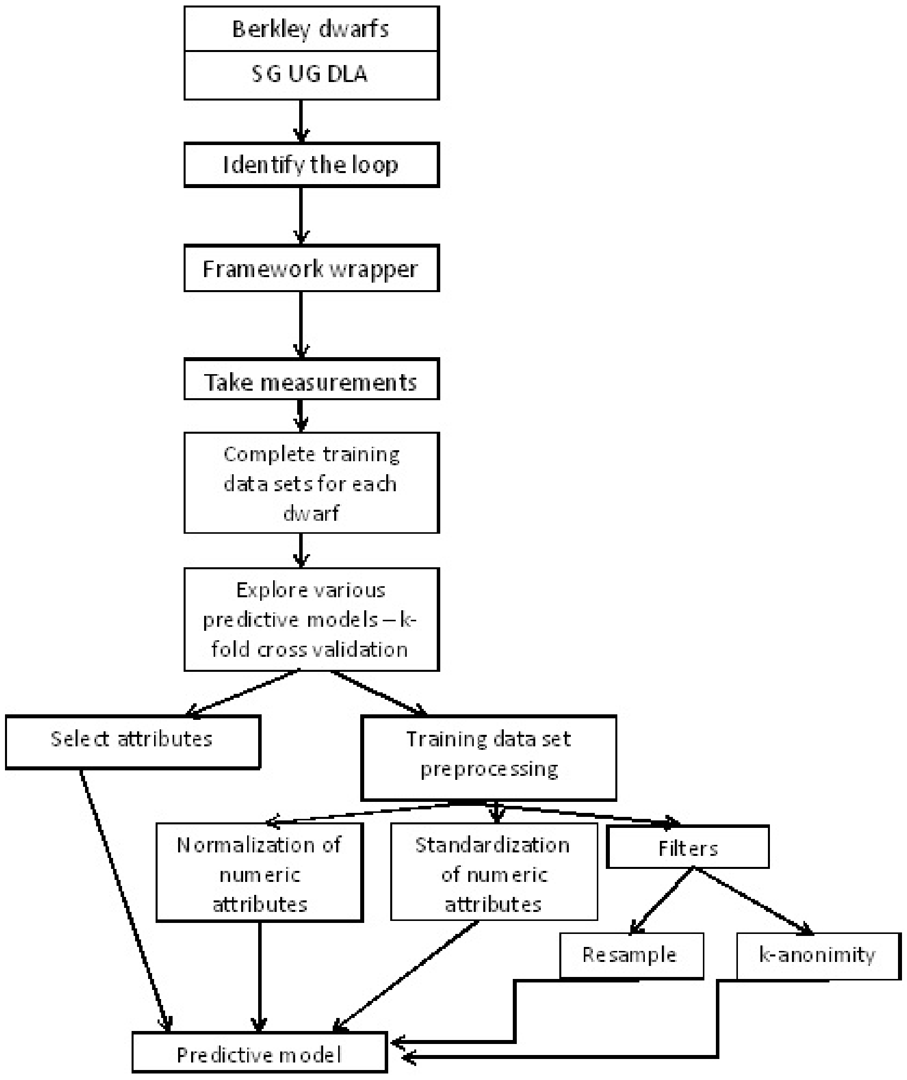

Berkley dwarfs
SG UG DLA
Identify the loop
Framework wrapper
Take measurements
Complete training data sets for each dwarf
Explore various predictive models – k-fold cross validation
Select attributes
Training data set preprocessing
Normalization of numeric attributes
Standardization of numeric attributes
Filters
Resample
k-anonimity
Predictive model

Figure 4.1: Diagram of our approach

4.2 Implementation Details

The section presented below will go through all the implementation details and steps that we take.

Our plan is as follows:

1. Select a loop for which we would want to gather information to build/test our model;
2. Wrap the loop with developed framework in order to gather performance measurements;
3. Gather performance information;
4. Complete the data set which is going to be used by the model to learn;
5. Build and evaluate the model;
6. Explore various preprocess techniques to improve a model's accuracy, 5.5.

4.2.1 Target Loop Selection

As we mention earlier in this work (3.5) we picked three computation patterns for performance prediction:

- Structured Grid;
- Unstructured Grid;

- Dense Linear Algebra.

In a given application we analyze all the loops using *gprof* profiler (3.8)to find the most computationally intensive loops. We use *line-by-line* profiling mode in order to find which lines of the code consume a considerable percentage of the total running.We are interested in computationally expensive loops because in this case choosing a beneficial combination of scheduling policy and a chunk size affects the performance of the whole application. After identifying those parts we analyze the loops which include those lines: we are picking the loops where computations meets the requirements of a particular dwarf. Code snippets pertaining to a particular computation pattern employ similar memory access pattern and resource management.

In this subsection we would like to give examples of loops in the applications we use and explain why the computation inside the loop can be rated as an example of a dwarf.

1. *Structured Grids.*

   The computations which belong to this pattern are applied to a regular multidimensional grid with a well-defined structure. The operations inside the computation represent a number of grid update steps. During each of the steps the values of a current point is updated using the values of its close neighbors. Usually, the applications working on images are put into the Structured Grid category. The reason for this is that images are regular grids on practice. Applications which traverse and use different parts of an image are also categorized as Structured Grids.

To exemplify our way of reasoning we would like to present a reader a snippet of code which we found to be the most computationally intensive and the closest to the pattern in comparison with other loops presented in the code. The application that we are using to give an example is SRAD, one of the programs from the Rodinia benchmark suite ([17]).

After profiling is done, a line-by-line analysis detects the most computationally intensive parts of the code.

```
time(%) name
9.75 main.omp_fn.0 (srad.cpp:153 @ 401cbf)
8.43 main.omp_fn.0 (srad.cpp:143 @ 401ad5)
```

Our next step is to investigate the loops which have those lines in order to check whether it exhibits SG pattern. The body of the loop that contains grid computation looks as follows:

```
...
for (int i = 0 ; i < rows ; i++)
{
for (int j = 0; j < cols; j++)
{
...
dN[k] = J[iN[i] * cols + j] - Jc;
dS[k] = J[iS[i] * cols + j] - Jc;
dW[k] = J[i * cols + jW[j]] - Jc;
dE[k] = J[i * cols + jE[j]] - Jc;
G2 = (dN[k]*dN[k] + dS[k]*dS[k] \
+ dW[k]*dW[k] + dE[k]*dE[k]) / (Jc*Jc); \\line # 143
...
c[k] = 1.0 / (1.0+den) ; \\ line # 153
...
}
}
...
```

In this code *J* variable represents an image, *rows, cols* represent problem size. As we can see, the image which is a grid is heavily used, its parts are accessed in a pattern which depends on the values of several values.

As the image's parts are visited, we would want to look at how the cache is utilized as such data structures are large in size and cannot be fit into cache memory. The applications pertaining to the dwarf should have similarities in the way they optimized cache usage, and how the number of cache hits and misses fluctuates. The large size of an image leads to an importance of scheduling the iterations among the threads. In terms of the chunk size to use, it has to be small enough to exploit parallelism and large enough to utilize the resources of each processing unit;

2. *Unstructured Grids.* This pattern usually deals with situations when there

is no direct mapping between where things are in memory and how they are represented in the physical space. The computations are performed on an irregular mesh or grid. An ideal example of the pattern would be a computation which tries to update mesh elements using information from its neighbors.

For the explanatory purposes we would like to present the following snippet of code taken from CFD solver application from the Rodinia benchmark suite ([17]). The application is an unstructured grid finite volume solver for the 3D Euler equations.

Line-by-line analysis output is:

```
time(%) name
22.43 sqrt (cmath:429 @ 40205c)
9.02 sqrt (cmath:429 @ 401f8e)
5.53 compute_speed_sqd (euler3d_cpu.cpp:148 @ 402029)
4.41 compute_velocity (euler3d_cpu.cpp:143 @ 40200d)
3.98 compute_flux (euler3d_cpu.cpp:245 @ 401fca)
```

Lines number 429, 148, 143 are not in the loop bodies, next computationally intensive line is number 245 is found in a UG loop:

```
for(int i = 0; i < nelr; i++)
{
...
momentum_i.x = variables[NVAR*i + (VAR_MOMENTUM+0)];
momentum_i.y = variables[NVAR*i + (VAR_MOMENTUM+1)];
momentum_i.z = variables[NVAR*i + (VAR_MOMENTUM+2)];
float density_energy_i = variables[NVAR*i + \
VAR_DENSITY_ENERGY];
...
momentum_nb.x = variables[nb*NVAR +\
(VAR_MOMENTUM+0)]; \\line # 245
...
}
```

The object *variable* represents an irregular mesh which is processed in different stages of the computation. As we can see, the unstructured grid presented by *variable* is heavily visited. The mesh is irregular, thus the index of a point which needs to be accessed is carefully calculated using momentums for each of the dimensions of the grid. With respect to OpenMP scheduling, the mesh partitions should be distributed among threads in such a way that the next partition to be processed is close in memory to a previously accessed one. Such application may have similarities in cache utilization;

3. *Dense Linear Algebra.* This computational pattern represents classical vector and matrix operations. The data is usually processed in a form of rows, columns or matrix blocks.

   To exemplify our way of thinking when choosing a loop we would like to present a piece of code taken from LUD application from Rodinia benchmark suite ([17]). The program calculates a set of linear equations.

The most computationally intensive lines of the code are:

```
time(%) name
58.75 lud_omp (lud_omp.c:30 @ 400f4f)
34.56 lud_omp (lud_omp.c:38 @ 400fb7)
```

After the lines are identified we investigate the loops which contained them.

```
...
for (j=i; j <size; j++)
{
sum=a[i*size+j];
for (k=0; k<i; k++)
{
sum -= a[i*size+k]*a[k*size+j]; //line # 30
}
a[i*size+j]=sum;
}

...
```

In this snippet of code a variable *a* is a matrix which is processed for both reading and writing purposes. Applications in this dwarf have similar memory hierarchy issues - distribution of data among the L1 - L3 caches in such a way that the most commonly used parts of vectors/matrices are in cache; since the computation overlaps such applications prevent them from having a good level of scalability. The latter factor leads to the fact that with the increase of the number of threads we use the performance gain is not very significant after some point.

Since line number 30 is in the DLA loop already, we do not explore line number 38.

To summarize, we try to pick the loops which include as many features of a corresponding dwarf as possible, another factor which influence our choice - problem size, working with a particular dwarf we picked loops which have the same upper bound.

4.2.2 Measurements Framework Introduction

To build a predictive model we need to gather a lot of parameters of the program. In order to do that we develop a framework which gathers information about all necessary parameters; later they are used to form a training data set for the model. The framework allows taking measurements of hardware counters and execution time. The measurements are taken separately so there are no conflicts on a low hardware level between the parts which are involved in measurements.

Our framework consists of three parts:

1. Initialization - all the variables are set up in this section. The code involved in this part is placed right before the loop being tested. PAPI library is initialized in this part as well;

2. Start - the measurements are started up here. The part is placed right before the loop being tested and right after the Initialization part;
3. Finalization - all the measurements are finalized and are printed in a CSV (Comma Separated Values) format.

We developed a framework consisting of three parts because of several factors:

- All necessary variables and PAPI library have to be initialized once in a program. That is why this part is placed before the parallelized loop (outside the parallel region);
- Since we take hardware counters measurement on a per-thread basis the start part is placed inside the parallel region - to capture measurements from all participating threads;
- To finalize (stop counting the events) the measurement for each thread we put the en part right before the exit from a parallel region.

The three parts are implemented in a form of header files which are included in the code being tested using an *#include* directive. In other words, the frameworks "wraps" the loop that we want to get performance information about.

We implement our framework in a form of header files because it allows us to instrument the parallel region itself. In a sense, the framework wrapper is a part of the program, this fact reduces the overhead as compared to calling external functions. The latter would not allow us to operate inside the parallel region before the loop is being parallelized.

Further we would like to give a detailed description of each of the framework's parts.

4.2.3 Framework Implementation Details

We would want to start the detailed description of the framework with the initialization part.

As mentioned earlier in this work, we are interested in both execution time measurements and hardware counters measurements. These gatherings are performed separately: PAPI hardware counters measurements would incur a certain degree of overhead. We would not want to include this overhead in execution time measurements since it can affect the choice of a suboptimal combination of scheduling policy and a chunk size - the hardware counters are captured inside the parallel region, thus the time spent of evoking PAPI functions will be included in the loop's execution time. Another reason for separate measurements is that we measure counters in groups (discussed in the next couple of paragraphs), hence we run the application each time we want to gather information on a particular group; event sets can include different number of counters, thus the overhead incurred by capturing a particular group of counters is not constant from one even set to another.

We take measurements of two different aspects of a program - execution time and hardware counters. Each of those aspects requires separate operations taken to set up the measurements.

- *Execution time.* This part of the measurements tracks execution time when various combinations of OpenMP scheduling policies and chunk sizes are used. Thus, in an Initialization part we have to set up the chunk sizes that are going to be used. Various chunk sizes are passed through a text file which looks as follows:

```
2  0
5  0
10  0
5  0
10  0
```

Each of the file's lines stand for the percentage of the problem size which is going to be used as a chunk size, and the mode 0 signifies that we are measuring execution time. The percentage of the problem size is chosen as a chunk size because it allows to adjust the number of iterations each thread gets, if a chunk size is a fixed number than for big problem sizes the loop parallelization is very fine grained. The percentage of the problem size as a chunk size enables us to reflect the magnitude of the problem size.

The input in a form of a text file is chosen because we have to run the application for different scheduling policies and chunk sizes. Since both scheduling policy and a chunk size are a part of OpenMP API, they have to be hardcoded, i.e. we can't pass them via a command line. To make the measurement process more efficient and reduce the number of executable files we have to run, we hardcode the policy and a chunk size is changed from one run to another - a file line number is passed to an application via command line interface.

In the Initialization part for execution time measurement we indicate which file contains chunk sizes percentages and we read it line by line into two variables - percentage of a problem size which is going to be used as a chunk size; measurements mode. The line of the file which stands for a particular percentage of problem size that we want to try is passed to the application being tested trough a command line;

- *Hardware counters.* This part of measuring process is responsible for monitoring the values of all the hardware counters available on the platform. Initially we have no knowledge about which CPU signals counts (events) are important in terms of OpenMP scheduling policy selection; we don't have the information on which hardware counters reflect the way the workload is distributed among threads. That leads us to measuring all the counters available on the platform. To make this process meaningful and efficient we measure counters in groups.

  The groups completion process is very important because hardware counters are measured on a very low level, thus, we have to make sure that each part of the CPU is measured separately. This condition will eliminate an overlapping between measurements, making them more accurate and trustworthy.

  First of all, we find out which counters can be measured together.

  We take a list of all the counters available on the platform using *papi_avail* PAPI utility . The command outputs all the counters available and indicates whether a particular counter is native (events native for a certain platform) or derived (calculated using native counters) (A):

(...)

PAPI_L1_DCM - not derived, Level 1 data cache misses

PAPI_L1_ICM - not derived, Level 1 instruction cache misses

PAPI_L2_DCM - not derived, Level 2 data cache misses

PAPI_L2_ICM - not derived, Level 2 instruction cache misses

PAPI_FP_OPS - not derived, Floating point operations; optimized to count scaled single precision vector operations

PAPI_FAD_INS - not derived, Floating point add instructions (Also includes subtract instructions)

(...)

After having all the counters available in the list we combine them in groups in the following manner: take the first counter in the group, put in a list, add the second counter to the group, use *papi_event_choser* . This command checks whether the supplied counters can be measured together. For example:

```
papi_event_chooser PAPI_L1_DCM PAPI_FP_OPS PAPI_FAD_INS PAPI_L1_TCH
```

will produce

Event PAPI_L1_TCH can't be counted with others

Counters PAPI_L1_DCM PAPI_FP_OPS PAPI_FAD_INS are not derived counters, PAPI_L1_TCH is derived. To find out which native counter is used to calculate PAPI_L1_TCH we ran

```
papi_avail -e PAPI_L1_TCH
```

The output:

```
...
Native Code[2]: 0x40000011 |DATA_CACHE_MISSES|
Number of Register Values: 2
Register[ 0]: 0x0000000f |Event Selector|
Register[ 1]: 0x00000041 |Event Code|
Native Event Description: |Data Cache Misses|
...
```

As we can see DATA_CACHE_MISSES (DCM) event is used. Since PAPI_L1_TCH and PAPI_L1_DCM are related to L1 cache, they can't be gather together because the value of PAPI_L1_DCM is used to calculate PAPI_L1_TCH. As the values of the events counts are saved in special registers (counters) the same register is going to be used to be both written (to record PAPI_L1_DCM) and read (to calculate PAPI_L1_TCH) at the same time, that leads to incompatibility of some derived and native counters.

We keep adding counters from the initial list until a newly added counter can't be measured together with the counters which are already in the list. After that a newly picked counter becomes the first in our algorithm and a new group is formed. We form 16 groups. We understand that various permutations of the counters would end up in fewer groups, however, the number of groups does not affect the measurement accuracy - measurements for each group are performed in separate runs. The number of counter in the group also does not affect the accuracy of measurement - since counters for a group are compatible with each other, they do not interfere while application's execution.

Since each group must be measured in a separate application run we have to execute the program for each of the group. The number of the group is passed in a way similar to the chunk size - through a text file.

We are able to form 16 groups of counters. To indicate which group should be used at the moment we use a text file consisting of the following information:

```
0  1
1  1
2  1
...
15  1
```

The first number in each line represents the group number which has to be taken care of; another number shows the measurements mode - hardware counters. We have to have two values for the mode to be able to use the framework for both execution time and hardware counters. Only the line number is passed to the benchmark program.

Each line of the file is read into two variables - group id and measurement mode.

The following snippets of code exemplifies the way the groups are formed in the initialization part.

First we hardcode the number of counters in each group:

In the header file which is responsible for the Initialization part we first specify the number of counters per group:

```
(...)
switch (gr_id)
{
case 0:

NUM_EVENTS=3;
break;

case 1:
NUM_EVENTS=2;
break;

case 2:
NUM_EVENTS=3;
break;

(...)
}
(...)
```

The next step we should take is forming the groups of the counters based the group id passed to an application:

```
(...)
switch (gr_id)
```

```
{
case 0:
group1[0]=PAPI_L1_DCM;
group1[1]=PAPI_L1_ICM;
group1[2]=PAPI_L2_DCM;
NUM_EVENTS=3;
break;

case 1:
group1[0]=PAPI_L2_ICM;
group1[1]=PAPI_L1_TCM;
NUM_EVENTS=2;
break;

case 2:
group1[0]=PAPI_L2_TCM;
group1[1]=PAPI_FPU_IDL;
group1[2]=PAPI_TLB_DM;
NUM_EVENTS=3;
break;
(...)
```

As we are dealing with PAPI hardware counters interface (3.6) we have to set up a PAPI library to be able to use the interface. The library initialization is performed using the following function:

```
(...)
PAPI_library_init( PAPI_VER_CURRENT )
(...)
```

Since we are working on a multi-threaded applications, we have to initialize the library support for each thread:

```
(..)
PAPI_thread_init((unsigned long ( * )( void ) )omp_get_thread_num))
(...)
```

The latter setup allow each thread to call all the PAPI functions separately, thus we can take measure nets for each thread and after the data is gathered we can find an average of a particular value to get a general view across all the threads. We would want to take hardware counters on a per-thread basis in order to generalize the measurements across the threads. If we capture counters outside the parallel region, only one thread (a master thread ([8]) performs events counts and information from other threads is missing, it can lead to an improper characterization of the loop.

After all the necessary variables are initialized and set up we can start the measurement process. The Start part of the framework focuses on two aspects: execution time measurements and hardware counters measurements.

- *Execution time measurements.* Based on the mode we use (0 for execution time measurements) we initialize loop execution time counting:

```
(...)
start_usec = PAPI_get_real_usec();
(...)
```

Function PAPI_get_real_usec() ([14]) returns total real time elapsed since some starting point, the time is measured in microseconds. Only a master thread takes this measurement as we need an overall execution time for the loop - a metric used to evaluate the performance of a snippet of code.

- *Hardware counters measurements.* For the measurement mode equal to 1 we perform hardware counters measurements. For that purpose we use the following sequence of steps *inside* the parallel region:

  1. PAPI_create_eventset(&private_event_set) - create an EventSet handler, we need it to be able to reach the group of the counters;
  2. PAPI_add_events(private_event_set, (int *) group1,NUM_EVENTS)) - add events which are specified by a current group (whose number is passed to an application) to the created event set;
  3. PAPI_start(private_event_set)) != PAPI_OK) - starts counting the values of the hardware counters which comprise the specified group or event set.

Those three steps described above are taken inside the parallel region. The reason for this is that we want to collect the hardware counters' measurements performed by each thread. If those operations are done outside the parallel region, all the counters will be measured by master thread only, thus not including measurements inside the parallel region we lose information from other threads.

The purpose of the Start part of the framework is to launch the measurements.

After the measurements are initiated, the loop is being executed as it normally would. After all the computations inside the loop are done we have to stop measurements of execution time and hardware counters. Since we perform execution time and counters measurements separately, we would want to describe the end part of the framework for each mode:

- *Execution time.* We take another time stamp outside the parallel region after the loop is executed using:

  ```
  end_usec = PAPI_get_real_usec();
  ```

  and subtract it from the initial time stamp we got in the Start part.

- *Hardware counters.* First of all, we stop all the counters measurements made by each thread using:

  ```
  PAPI_stop( private_event_set , values );
  ```

  All the counters measured by a thread are copied into *values* array for further processing. After each thread *values* array having measured counters we find an average of each of the counters across all the threads. The average values of hardware counters enable us to get a generalized view of low-level loop's behavior.

The source code of all the parts of the framework can be found in B.

In this section we presented our methodology and technique to take performance measurements on the code. Further we are going to discuss our experiment and findings.

Chapter 5

Experiments and Findings

In Chapter 5 we are going to introduce the experimental part of our work. We will present execution time and hardware counters measurements, machine learning models we use along with the explanation of our findings. Also, the steps we take to investigate to built models are described.

5.1 Testbed

The following section presents the test bed on which we run our experiments.

The executable files are run on Shark cluster, nodes shark25 through shark29 ([18]).

Each of the nodes mentioned is based on SUN X2200 Server.

The CPU that are used for our experiments have two 2.2 GHz quad core AMD

Opteron cores, 8 cores total per node. The capacity of main memory is 8 Gb.

The network involved includes a 96 port 4xInfiniBand SDR switch and 48 port Linksys GE switch.

5.2 Benchmarks Considered

This section presents the list of benchmarks we use for our experiments to gather training data for a predictive model. Only the OpenMP version of the applications indicated are used. Each application being tested pertains to one of three dwarfs mentioned earlier in 3.5.

1. *Rodinia Benhcmark suite*, [17].

    - *Streamcluster* . The application pertains to a Dense Linear Algebra Dwarf. The application is a modification of the streamcluster benchmark in the Parsec suite developed by Princeton University. For an input matrix it finds a predefined number of medians in such a way that each point is assigned to a closest center. The sum of distances to the power of 2 defines the quality of clustering;
    - *LU Decomposition* . The application pertains to a Dense Linear Algebra dwarf. The algorithm used in this application calculates solution for a set of linear equations. It decomposes the input matrix into a product of a lower triangular matrix and an upper triangular matrix;
    - *Back Propagation* . The application is related to an Unstructured Grid

dwarf. It implements a machine learning algorithm which calculates weights for nodes in a layered neural network. There are two phases in the application: the Forward Phase, which propagates activations from an input layer to an output one; and Backward Phase in which an the observed error and the requested value on the output layer if propagated back to an input layer to adjust weights;

- *CFD Solver* . The application is categorized as Unstructured Grid. It solves the three-dimensional Euler equations for compressible flow;
- *Myocyte* . The application pertains to a Structured Grid dwarf. It simulates cardiac heart muscle cell;
- *Particle Filter* . The benchmark is related to a Structured Grid dwarf. The algorithm implemented within the framework of the application estimates the location of a target provided with noisy measurements of that location and an a path in a Bayesian framework;
- *SRAD* . The application pertains to a Structured Grid dwarf. Based on partial differential equations (PDEs) it presents a diffusion method for ultrasonic and radar imaging applications. The most frequent use of this algorithm is removing locally correlated noisy without affecting important features of the image;
- *Hotspot* . The application is categorized as a Structured Grid. It is a tool used for estimation of temperate of a processor based on architecture and power measurements. The algorithm employed in the benchmarks solves a series of differential equations for block;

- *Kmeans* . The benchmark is related to a Dense Linear Algebra dwarf. Implements a clustering algorithm widely used in data-mining methodology;

2. *Parboil Benchmark suite*, [19].

    - *Stencil* . The application is related to a Structured Grid dwarf. It presents an iterative Jacobi stencil operation on a regular grid;

3. *Traditional matrix-matrix multiplication* . The application pertains to a Dense Linear Algebra dwarf. This benchmark performs a dense matrix multiplication operation.

5.3 Machine Learning Models Employed

This section is aimed at giving a detailed description of machine learning models we try in our experiments. Earlier in this work (3.2) we gave a brief introduction to types of the models.

5.3.1 Supervised Machine Learning Task Type

Since we categorize all the runtime configurations into classes (each class represents a combination of an OpenMP scheduling policy and a chunk size) the most suitable type of machine learning task would be a supervised one. This type of a model infers a classification function from a labeled training data set. The training data set consists of *training examples*. Each of these examples is represented by of a set of input values and an output value - the class the example is related to. A supervised learning algorithm analyzes training set and produces a *classifier* - a classification function having a set of values on the input outputs a predicted class.

We take the following steps to solve a supervised machine learning problem:

1. Categorize all the examples, which form a training data set. In our case, we classify a runtime configuration into such a class (a combination of an OpenMP scheduling policy and a chunk size) where the execution time achieved is the smallest - we estimate execution time for each of the combinations.

   Example: static OpenMP scheduling scheme along with chunk size equal to 5% of a problem size enabled to minimize execution time (compared to the execution time we get from other combinations) for a particular runtime configuration (a compiler we use, number of threads, problem size, hardware counters values); thus, the configuration is labeled as ”s_5”;

2. Gather training data set. In order to do so, we take measurements of execution time for various combinations of an OpenMP scheduling policy and a chunk size; hardware counters are measured for all runtime configurations when no scheduling is applied. We need to gather values of the counters only for the default scheduling since the default program's behavior for characterization purposes. Default scheduling implies static scheme and a chunk size equal to the number of iterations divided over the number of threads. The latter approach allows getting a generic picture on an application's behavior;

3. Determine input features which are to represent the learned function. Initially, we have all 40 measured hardware counters, a compiler used to produce an executable file, the number of threads, and execution time when no policy is applied as those features. Later we select attributes based on how well they separate the training data set into classes. We use the WEKA software tool for that purpose ([20]);

4. Run the learning algorithm on a training data set to train a model;

5. Evaluate the model by supplying a test example which is not included in a training data set. The model has to predict the class for which the example should be related.

In our work to complete steps 4 and 5 we use k-fold cross validation. The algorithm of model evaluation works as follows:

1. Break the training data set into k groups;

2. Train the models using k-1 groups as a training data set;

3. Classify examples included in k-th group are classified using the built model. All combinations of the groups are tried and the average result is considered. We employ the WEKA software tool to perform k-fold cross validation.

5.3.2 Supervised Machine Learning Algorithms

The section below gives a minute description of the machine learning algorithms that we use to build and validate our predictive model. We attempt to test several models. The choice of models is explained by the fact that they are widely used algorithms in a scientific community and well-known to produce stable and accurate predictions ([10]).

- *Bayesian Network (BN).* Also known as belief networks, BN are referred to a family of probabilistic graphical models (GMs). Each node of the graph represents a random value and edges represent probabilistic dependencies between corresponding random values. In BN, each variable is independent of the nodes which are not its descendants provided that that the state of the parents is known. The graph structure which is often used is a directed acyclic graph (DAG). For example, a BN can represent e relationship between diseases and symptoms. Given symptoms, the networks computes the probability of a patient having a particular disease.

  To provide a reader with a better used of how BN actually works we would like to exemplify the algorithm as follows ([21]).

  Suppose we have two events which can turn the alarm (A) on - coolant pipe leakage (CP) and high temperature (HT). The DAG below shows the relationship between those three variables:

  The probabilities of various situations in this model are:

  The joint probability function is:

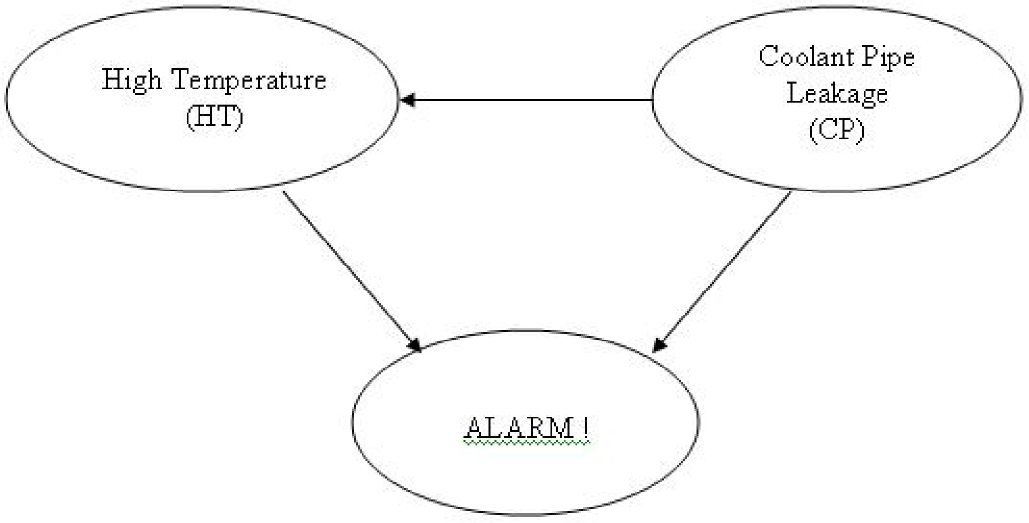


Figure 5.1: DAG, BN example

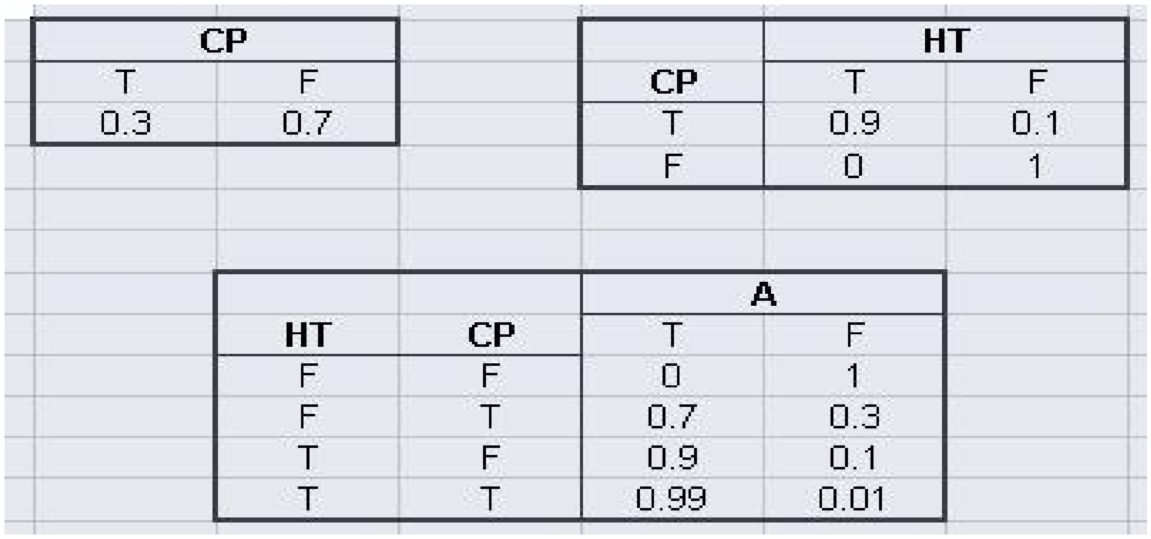
P(A,HT,CP) = P(A — HT,CP)P(HT — CP)P(CP)

Now we can infer probability of any situation we want, in this example the probability of the situation when the temperature is high and the alarm is on is:

- *Naive Bayes.* In simple words, the classifier assumes that there is no relationship between presence or absence of one set of an object's features and presence or absence of another set of features. The probabilistic model of the algorithm is a conditional model:

  p(C —F1, ..., Fn)

  C - represents a class, F1 through F n - features of objects to be classified, those characteristics are used to separate the training set into classes. Using Bayes' theorem we can present the probability as:



CP	
T	F
0.3	0.7

	HT	
CP	T	F
T	0.9	0.1
F	0	1

		A	
HT	CP	T	F
F	F	0	1
F	T	0.7	0.3
T	F	0.9	0.1
T	T	0.99	0.01

$$\mathrm{P}(CP = T \mid A = T) = \frac{\mathrm{P}(A = T, CP = T)}{\mathrm{P}(A = T)} = \frac{\sum_{HT \in \{T,F\}} \mathrm{P}(A = T, HT, CP = T)}{\sum_{HT, CP \in \{T,F\}} \mathrm{P}(A = T, HT, CP)}$$

- *Support Vector Machine (SVM)* . The algorithm constructs a hyper plane or a set of hyper planes in high-dimensional space. The larger the distance between the hyper plane and a data point in a training data set the better the model is. To be able to classify the sets of data points which are not linearly separable the original space is mapped into a much higher-dimensional space;

- *Artificial Neural Networks (ANN)*. The network consists of a group of artificial neurons. A neural network is an adaptive system. It changes its structure during the learning process. The neurons in the network are divided into groups - layers. The very first layer has input nodes, inner layers that process input information, and output layers that represent the classes. Learning an artificial neural network boils down to evolving the neurons by continuous evaluation of their output, calculating the weighted sum and comparing the result with

$$p(C|F_1,\dots,F_n) = \frac{p(C)\ p(F_1,\dots,F_n|C)}{p(F_1,\dots,F_n)}.$$

a threshold. The most frequently used algorithm for ANN training is back-propagation of error. Initially, each input node is assigned an arbitrary weight. After calculating, the error between actual and desired outputs is propagated backwards to adjust the weight. The process continues until the error does not exceed a threshold;

- *Random Forest.* The algorithm constructs decision trees and outputs a class which is a mode of the classes output by individual trees. The term of a decision tree is central in the algorithm being discussed. The goal is to create a model to predict the value of a target variable having several input values. An interior node of the tree represents an input value, leaves - a value of a variable to be predicted. A tree can be trained by dividing the training set into subsets in accordance with the attributes. The process is stopped when further splitting does not give better predictions. Thus, Random Forest algorithm is much more powerful than an individual decision tree. It can deal with a great number of input variables; the model can classify unlabeled data. When all the trees comprising the forest are built, all of the data is processed by the tree proximities are calculated for each pair of cases. If two cases reside in the same terminal node (a leaf) their proximity is increased. After that, the proximities are normalized by dividing over the number of trees. Proximities are used for replacing missing data and detecting outliers.

The algorithms presented are a part of the WEKA machine learning software tool ([20]).

5.4 Code Instrumentation And Training Data Set Completion

The following section presents a detailed explanation of our approach to building a predictive model which takes a set of parameters on the input (1.2) and produces a combination of OpenMP scheduling policy and a chunk size that allows for better performance on a certain platform. We will discuss which steps we took to complete the training set, machine learning models we tested, training data set preprocess methods we employed, and collector API involvement.

5.4.1 Code Instrumentation

As we mentioned earlier in this work we have to gather performance data to build a predictive model. In order to complete our training data sets (one set per dwarf) we develop a framework for taking all necessary measurements (4.2.2). After identifying the loop that consists of computations pertaining to a certain dwarf (4.2.1) we wrap the loop with the three parts of our framework. To exemplify the loop code instrumentation we would want to give a snippet of instrumented code from *streamcluster* application, Dense Linear Algebra dwarf:

- First of all, we read the file line from the command line:

  ```
  ...
  int file_line;
  file_line=atoi(argv[argc -1]);
  ...
  ```

  If the loop being tested is located in a function other than main(), we pass the variable to a particular function.

- Depending on what kind of measurements we take - execution time or hardware counters - we change the init.h file respectively:

  ```
  ...
  static const char filename[] = "time_data.txt";
  ...

  or
  ...
  static const char filename[] = "counters_data.txt";
  ...
  ```

- We wrap the loop with our framework:

  ```
  int size;
  #include"init.h"
  size=round(k2*chunk_size);
  if(size==0) size=1;
  ```

```
#include "start.h"
#pragma omp for schedule(guided,size)
for ( int i = k1; i < k2; i++ )
{
bool close_center = \
gl_lower[center_table[points->p[i].assign]] > 0 ;
if ( switch_membership[i] || close_center )
{
points->p[i].cost = points->p[i].weight *dist(points->p[i], \
points->p[x], points->dim);
points->p[i].assign = x;
}
}

#include "end.h"
```

- Variables, which a indicated as shared, private, firstprivate, etc. are moved into start.h header file which includes *#pragma omp parallel* construct. We take this step in order to take our hardware counter measurements for each thread participating in a parallel region;
- Next we compile the application:
  - For gathering data for hardware counters values we compile the application with each of the compilers;

– For gathering data for execution time we apply different scheduling policies (static, dynamic, guided) including the case when no policy is used and compile each version of the application with all each of the compilers.

5.4.2 Training Data Set Completion

In this subsection we are going to present all the steps we took to form a training data set for our model. Since the applications we explore pertain to three different computation patterns (3.5) we are going to build a model for each of the dwarfs. The training sets for each of the models are comprised of the performance measurements taken for applications belonging to the dwarf. Since the steps on training data set completion are the same for all dwarfs, we will describe the approach in general. Nevertheless, a model produced is unique for each dwarf, that is why its description will be given on a per-dwarf basis.

5.4.2.1 Execution Time and Hardware Counters Measurements Processing

After producing all necessary executable files for each of the applications we test we start taking measurements by running each executable file with the following runtime parameters:

- Different number of threads: 2, 4, 8;
- Different problem size. For SG dwarf - 64,512,1024; for UG dwarf - 97000,

193000, 200000; for DLA dwarf - 64, 512, 256204, 204800, 494020, 819000;

- Different compilers which are used to compile the code - GNY, PGI and UH compilers.

In order to make the measurement process efficient and interpretable we implement a script which runs our applications with all the parameters and prints out the results (execution time and hardware counters) in a Comma Separated Values (CSV) format; the following snippets of code present such script for execution time measurements:

```
#!/bin/sh
echo ",,,,2,2,2,5,5,5,10,10,10"
for compiler in gcc pgcc opencc; do
for problem_size in 97000 193000 200000; do
for num_threads in 2 4 8; do
export OMP_NUM_THREADS=${num_threads}
#echo "PLAIN"
echo -n "${compiler},"
echo -n "${problem_size},"
echo -n "${num_threads},"
#plain
./${compiler}_plain_backprop ${problem_size} -l 0
echo -n ","
for gr_id in 0 1 2; do
```

```
#"STATIC"
./${compiler}_static_backprop ${problem_size} -l $gr_id
echo -n ","
#echo "DYNAMIC"
./${compiler}_dynamic_backprop ${problem_size} -l $gr_id
echo -n ","
#echo "GUIDED"
./${compiler}_guided_backprop ${problem_size} -l $gr_id
echo -n ","
done
echo ""
done
done
done
```

and hardware counters measurements:

```
#!/bin/sh
for compiler in gcc pgcc opencc; do
for problem_size in 97000 193000 200000; do
for num_threads in 2 4 8; do
echo -n "${compiler},"
echo -n "${problem_size},"
echo -n "${num_threads},"
export OMP_NUM_THREADS=${num_threads}
```

```
for gr_id in 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15; do
./${compiler}_backprop ${problem_size} -l $gr_id
done
echo ""
done
done
done
```

Backpropagation benchmark (UG dwarf) listings are presented above. The measurements are taken for 2, 4 and 8 threads. Measurements scripts for other benchmarks differ only in the problem sizes specific to a certain dwarf (5.4.2.1) and the way each program is run.

As stated earlier in this work, we compile several versions (one version per scheduling policy) of the code. When taking measurements for execution time we run the applications for 5 times. We do that in order to be able to identify outliers and use averaged values.

The output of both scripts is CSV (Comma Separated Values) friendly.

Now we would like to discuss the data we gathered.

Execution time measurements. As mentioned earlier, we compile different versions of the code - various compilers vs. different scheduling policies vs. several chunk sizes applied to the loop being tested. As a result, we got the execution time of the loop for all the cases discussed. An example of execution time gatherings is presented in tables 5.2, 5.3, 5.4. The rest of the execution measurements for other benchmarks are taken in a similar way;

Hardware counters measurements. Initially, we have no knowledge on which hardware counter could emphasize differences and similarities between the snippets of code belonging to the same class. That is why we gathered performance measurements for all hardware counters available on the platform (5.1).

In table 5.1 we present an example of hardware counters measurements. Several counters are indicated, however, we measure all counters available on the platform. Execution Time and Class columns are added to the hardware counters table to form a training set (5.4.2). The rest of the counters measurements are taken in a similar way.

As mentioned earlier in this section, execution time measurements are taken for 5 times. Our next step is to identify execution time outliers for each runtime configuration and each combination of scheduling policy and a chunk size we tried. It is of utmost importance to identify the outliers - the values which are markedly smaller or larger than other values. Since we categorize runtime configurations based on execution time we have to make sure that the outliers do not affect the class a particular configuration is referred to. For that purpose we take the following steps:

1. Our observations show that outliers are the values which are much greater than the minimum execution time measured for a particular runtime configuration and each combination of scheduling policy and a chunk size. Since we are interested in investigating cases where the execution time is close to the observed minimum, our first step is finding a minimum value for each runtime

Table 5.1: Hardware counters measurements (execution time and class columns added), kmeans, DLA dwarf

Compiler	Problem size	Number of threads	Execution time	PAPI_L1 _DCM	PAPI_L1 _ICM	PAPI_DP _OPS	Class
GNU	768	2	539.25	494	90	0	d_5
GNU	768	4	402.5	328	76	0	d_10
GNU	768	8	12247	192	92	0	d_10
GNU	1024	2	681.6	716	83	0	d_2
GNU	1024	4	512	350	78	0	g_2
GNU	1024	8	17683.5	197	89	0	d_10
GNU	1536	2	981.6	944	69	0	d_5
GNU	1536	4	658.75	496	63	0	d_5
GNU	1536	8	18173.5	271	80	0	g_10
PGI	768	2	579.4	540	92	384	d_5
PGI	768	4	299.8	389	99	192	d_2
PGI	768	8	178	190	97	96	d_5
PGI	1024	2	773.2	647	94	512	d_10
PGI	1024	4	375	506	107	256	d_2
PGI	1024	8	219.25	251	102	128	d_2
PGI	1536	2	1124.4	945	91	768	d_5
PGI	1536	4	557.6	690	112	384	p
PGI	1536	8	321.2	278	102	192	g_2
UH	768	2	499.3333	667	219	0	p
UH	768	4	446	562	239	0	s_5
UH	768	8	646.2	388	234	0	d_5
UH	1024	2	649.5	815	221	0	p
UH	1024	4	622	564	231	0	g_10
UH	1024	8	650	535	231	0	g_5
UH	1536	2	1001	1006	227	0	p
UH	1536	4	851.8	799	245	0	g_10
UH	1536	8	634.2	533	245	0	g_2

configuration and each combination of scheduling policy and a chunk size;

2. After the last step is completed, we still have 5 data sets with measured execution timings and 1 data set with minimum values for each runtime configuration and each combination of scheduling policy and a chunk size. Next, the heuristics that we employed considers those 5 data sets and removing its values which are less that 150% of the minimum value for a particular runtime configuration and each combination of scheduling policy and a chunk size;

3. A previous step produces 5 data sets which did not include outliers. The next operation performed is considering these 5 data sets and calculating average values for each runtime configuration and each combination of scheduling policy and a chunk size (5.2,5.3, 5.4);

The importance of finding and removing the outliers cannot be overestimated, since we classify each runtime configuration based on the execution time observed. An outlier can lead to a misclassification problem. The latter leads to wrong instance labeling (locating an instance into a class), which, in turn, prevents the model from reflecting a real relationship between the parameters on the input and the class predicted.

5.4.2.2 Training Data Set Building and Runtime Configurations Categorization

After outliers are removed from those 5 data sets for execution time measurements we finally have 1 data set containing timings averaged across all the 5 runs we performed for each runtime configuration and a combination of scheduling and a chunk size used. Now we would want to categorize each runtime configuration, hereinafter referred to as an ”instance” into a class. The latter means that we would want to find out, which combination of a scheduling policy and a chunk size allowed getting the shortest execution time. Since we have 4 scheduling policies (no policy, static, dynamic and guided) and 3 types of a chunk size (2, 5 or 10 percent of a problem size used for the loop), we have 10 classes. Notations for the classes are as follows:

- *p* - No scheduling policy is applied;
- *s_2, s_5, s_10* - Static scheduling policy is applied with a chunk size equal to 2%, 5% and 10% of the problem size respectively;
- *d_2, d_5, d_10* - Dynamic scheduling policy is applied with a chunk size equal to 2%, 5% and 10% of the problem size respectively;
- *g_2, g_5, g_10* - Guided scheduling policy is applied with a chunk size equal to 2%, 5% and 10% of the problem size respectively;

To categorize a particular instance we find a minimum execution time among different combinations of scheduling policies and chunk sizes.

Notation conventions for execution time measurements:

- *2, 5, 10* - Percentage of the problem size which is used as a chunk size;
- *plain/sts/gd* - different scheduling policies applied: no policy, static, dynamic, guided respectively;

- *cmplr* - Compiler which is used to compile the executable files;
- *n* - Problem size which is used;
- *thr* - Number of threads used to parallelize a program;
- Each cell under a particular scheduling policy represents execution time in microseconds;

An example of execution time measurements is presented in the tables 5.2,5.3,5.4. For the very first case (GNU, 512, 2), the minimum execution time observed is 15429333.4; therefore, the class into which we put the instance is ”g_10” (guided scheduling and chunk size equal to 10% problem size applied appeared to be the best option in terms of performance). We measured execution time of a serial version as well (5.5). We categorize all the instances for all the application we tested.

After each instance has been assigned a labeled class we can form a training set. The latter consists of input parameters for the model (4.1) and a class assigned to a particular instance. To build the training set for a particular application benchmark we insert a column named ”plain” into a table with hardware counters measured for that application. Finally, we insert the column with class labels for each instance as the last column into the table with hardware counters (5.1).

We use execution time measured for the case when no scheduling policy is applied because this parameter describes an application's default behavior.

After training data sets are completed for each application in a dwarf, we merge those into one training data set for a particular computation pattern. Since we have

Table 5.2: Execution time measurements, chunk size equal to 2% of a problem size, myocyte, SG dwarf

				2	2	2
cmpr	n	thr	plain (mcs)	static (mcs)	dynamic (mcs)	guided (mcs)
GNU	512	2	1.6E+07	15946604.6	15963762.4	15446702.6
GNU	512	4	8246914	8393010.4	8307207.4	8359042
GNU	512	8	4160176	4534380.2	4465514.6	4368096.8
GNU	768	2	2.4E+07	23864715.8	23309737.8	23366075.2
GNU	768	4	1.2E+07	12687833.8	12610999.4	12556642.2
GNU	768	8	6270507	6761903	6688692	6542805.2
GNU	1024	2	3.1E+07	31684850	31628977.2	31578117
GNU	1024	4	1.7E+07	16821098.2	16637834.2	16784462
GNU	1024	8	8301728	9271484.4	8963261.8	8773207
PGI	512	2	8858761	8965057.2	9104544.8	8785212.8
PGI	512	4	4843233	4847074	4929200.6	4800408.6
PGI	512	8	2691225	3272253.8	2980341.6	3084428.6
PGI	768	2	1.3E+07	13654660.6	13459269	13793344.4
PGI	768	4	7265812	7343777.6	7297896.6	7163391.8
PGI	768	8	4726506	4566504.25	4493513.25	4649898.5
PGI	1024	2	1.8E+07	17524532.6	18107586	17892690.2
PGI	1024	4	9520331	9808406.6	9714214.2	9732489
PGI	1024	8	6586447	6301184.8	5732790	5621263.8
UH	512	2	3E+07	30631190.2	30486354.4	30549600.2
UH	512	4	1.5E+07	15419131.4	15378307	15617422.6
UH	512	8	7502484	8163866.2	8160497.6	8063773.6
UH	768	2	4.5E+07	45883543.6	45714469.2	45743333.4
UH	768	4	2.3E+07	23094713.6	23087931.6	23441929.8
UH	768	8	1.1E+07	12235956.2	12227699.6	12249419.4
UH	1024	2	6E+07	61211531.2	60839244.6	61050670.4
UH	1024	4	3E+07	30951697.8	30798788.4	31274933.8
UH	1024	8	1.5E+07	16326667.8	16322200.6	16318787.8

Table 5.3: Execution time measurements, chunk size equal to 5% of a problem size, myocyte, SG dwarf

				5	5	5
cmpr	n	thr	plain (mcs)	static (mcs)	dynamic (mcs)	guided (mcs)
GNU	512	2	1.6E+07	15824277.8	15585004.8	15846547.4
GNU	512	4	8246914	8429731.6	8414151.4	8367647.4
GNU	512	8	4160176	5091372.8	5009307	4770194.4
GNU	768	2	2.4E+07	23505657.6	23159367	23490609.4
GNU	768	4	1.2E+07	12772716.2	12299544.4	12644464.2
GNU	768	8	6270507	7298329	7621563.8	7183134.6
GNU	1024	2	3.1E+07	31030417.8	31261857.2	31195414.4
GNU	1024	4	1.7E+07	16636997.8	16364144	16774311.4
GNU	1024	8	8301728	9919296.4	10029178.6	9537315.4
PGI	512	2	8858761	9025852	8876549.8	8944225.8
PGI	512	4	4843233	4930955.4	4829016	4903566.2
PGI	512	8	2691225	3145514.6	3098981.4	3002401.6
PGI	768	2	1.3E+07	13563380.4	13193141.6	13366439.4
PGI	768	4	7265812	7400219.6	7305039.4	7442591
PGI	768	8	4726506	4546107	4593212.2	4800426.75
PGI	1024	2	1.8E+07	17632352.8	17857791.2	17548393.4
PGI	1024	4	9520331	9597289	9712034.4	9680579
PGI	1024	8	6586447	6341709.4	6254563.8	6838215.8
UH	512	2	3E+07	30616044	30440657.2	30512420
UH	512	4	1.5E+07	15389248	15358002.4	15778427.2
UH	512	8	7502484	9063456.6	9056427.6	9040050.4
UH	768	2	4.5E+07	45722123	45435803.6	45502107.6
UH	768	4	2.3E+07	23300230	23097012.4	23811113.6
UH	768	8	1.1E+07	13246330	13233975	13256033.4
UH	1024	2	6E+07	60405837.4	60313075.2	60979818.4
UH	1024	4	3E+07	30613305.6	30255291.6	31592275.8
UH	1024	8	1.5E+07	17755022.2	17773459.4	17743447.8

Table 5.4: Execution time measurements, chunk size equal to 10% of a problem size, myocyte, SG dwarf

				10	10	10
cmpr	n	thr	plain (mcs)	static (mcs)	dynamic (mcs)	guided (mcs)
GNU	512	2	1.6E+07	15548351	15518860.2	15429333.4
GNU	512	4	8246914	9831200.8	9757491.8	9258118.8
GNU	512	8	4160176	6579725.8	6458427.4	6447418
GNU	768	2	2.4E+07	23439063.8	23432914.8	23612824
GNU	768	4	1.2E+07	14607336	14532873	13888211.8
GNU	768	8	6270507	10078117.4	9756309	9687849.2
GNU	1024	2	3.1E+07	31546803	31168364.8	31350769.6
GNU	1024	4	1.7E+07	19584326.2	19324861.8	18809610.4
GNU	1024	8	8301728	13066823.8	12868583.2	12786461.8
PGI	512	2	8858761	8969885.6	8763200	9106322.4
PGI	512	4	4843233	5688297	5553449	5389589.2
PGI	512	8	2691225	4297769.2	3935663.2	3694662.25
PGI	768	2	1.3E+07	13784260	13451074.2	13233047.8
PGI	768	4	7265812	8412534.6	8372293.4	8270512.8
PGI	768	8	4726506	6312023.2	5751190	5628811
PGI	1024	2	1.8E+07	18142212	17803955.4	18198845.2
PGI	1024	4	9520331	11371373.2	11081768	10904352.6
PGI	1024	8	6586447	8593125.5	7582240.8	7494236.8
UH	512	2	3E+07	30255935.4	30188385.4	31756624.2
UH	512	4	1.5E+07	18089759.6	17983703	17870938
UH	512	8	7502484	11718224	11679389.4	11680380.2
UH	768	2	4.5E+07	45206081.4	45154727	47703016.4
UH	768	4	2.3E+07	27147844	26941096	27018845.4
UH	768	8	1.1E+07	17627988.8	17644428.8	17576329.4
UH	1024	2	6E+07	60666037.2	60285461.6	63373135.8
UH	1024	4	3E+07	36001826.6	36024133.8	36016270.2
UH	1024	8	1.5E+07	23380448.8	23337766.2	23384271.6

Table 5.5: Execution time measurements for a serial version, myocyte, SG dwarf

cmpr	n	Serail execution time
GNU	512	30745333.2
	768	45971930.2
	1024	61266439
PGI	512	17133698.8
	768	25728623.2
	1024	34633382
UH	512	58836835.4
	768	87977450.2
	1024	117389900.6

3 dwarfs being tested, we get 3 training data sets which are going to be used to build a predictive model.

To further use the training set we have to present it using an ".arff" file format. Each instance is presented by a set of values of the attributes and an assigned class:

```
@relation SG
@attribute compiler {GNU,PGI,UH}
@attribute prob_size numeric
@attribute num_thr numeric
@attribute time numeric
@attribute PAPI_L1_DCM numeric
@attribute PAPI_L1_ICM numeric
@attribute PAPI_L2_DCM numeric
...
@attribute class {s_2 ,d_2 ,g_2 ,s_5 ,d_5 ,g_5 ,s_10 ,d_10 ,g_10 ,p}
```

```
@data
GNU,64,2,349,2935,99,510,..,d_2
...
```

5.5 Training Data Set Preprocessing and Experiment Analysis

In order to familiarize a reader with concepts, we use for the training data set preprocessing, we would want to give a brief overview of each of the techniques employed. We may want to use some of the preprocessing methodologies in order to improve our model accuracy. By "accuracy" we mean a percentage of instances which are classified correctly by the classifier (a machine learning algorithm).

1. *Attribute selection.* A process of identifying a subset of relevant model features. The need for this approach is justified by the fact that, for a certain model, some attributes can be redundant - they do not reflect differences between instances. The list of attribute selection advantages:

    - Model interpretability is better - features can be tailored to a predicted class, the relationship between instances of different classes can be inferred from the model's output;
    - The process of learning the model takes less time;
    - Possibility of a model to be over-fitted is reduced;

Attribute number selection is performed using Evolutionary Search. This algorithm performs a search process in order to find a good subset of attributes through evolving a population of individual attributes using generations;

2. *Numerical values normalization.* Normalizes numerical values in a training data set in such a way that all the values lie in the range 0 - 1.0. This methodology allows generalization across the data set;

3. *Numerical values standardization.* Standardize numerical values in such a way that the values have zero mean and unit variance. This approach might be useful because when k-fold cross validation is applied, we get two data sets - a training one and a test one. Standardization allows the same statistics to be applied to both data sets;

4. *Normalization of hardware counters values by the number of the total cycles of execution*, [7]. The technique is used to create event rates for each of the hardware counters;

5. *Gather data for more values of a problem size*;

6. *Add new applications to the dwarfs.* This approach, as well as the previous one, allows enlarging training data set, which, in turn, provides more information for the model;

7. *Reduce the number of classes.* Our observations showed that there are classes which are very close to each other. By "being close" we mean the fact that the set of attributes values pointing to the classes are pretty much the same. We

noticed that the difference between execution time (regardless of scheduling policy applied) for chunk size is equal to 5% of a problem size and that of chunk sizes 2% is pretty small (5-7% difference); a similar trend is observed for the difference between execution time taken for 5% and 10% chunk sizes. That led us to a conclusion that classes _5 are redundant. For example, for a model an instance of class s_2 and s_5 can be very similar in terms of values of the attributes and it may well put both instances into the same class. This fact may well lead to a misclassification, since we labeled all instances;

8. *Resample and Datafly filters applied to the training sets.* Resample filter produces a random subsample of the data set with replacement. The Datafly algorithm is based on k-anonymity approach ([20]). It anonymizes so-called "quasi-identifiers" - a number of attributes which can be used in a combination to create an almost unique key. A combination matches k classes, in our work k is equal to 1 since we want to distinguish each class individually. The Datafly algorithm uses domain generalization hierarchy function. This function presents the actual value of a numerical attribute in a less specific form of a range. Thus, an instance is classified based on a combination of attributes each of which is presented as a range. After attribute selection we have a number of features which present relevant information. However, this information may not be relevant to each of the classes; the selected attributes are relevant to all classes on average. The generalization that is provided by the Datafly algorithm leads to the fact that only the attributes relevant to a particular class are considered. This is the reason why generalized form of attributes (ranges)

does not harm precision of the model.

Now we would like to give a detailed description of the model exploration process for each of the dwarfs.

Each model is evaluated based on the accuracy achieved. Accuracy provides quantitative estimation of the model. Another metric for which we are going to use is false positive rate - the proportion of instances which are classified as class "x" (but pertaining to a different class) among all instances which are of class not "x". In order to provide a reader with a better understanding of FP (later referred to as FP) we would want to give the following example. Consider the following confusion matrix:

Predicted class		
class x	class y	Actual class
7	2	class x
3	4	class y

The matrix is interpreted as follows: 7 instances are predicted as class x and actually are of class x; 4 instances are predicted as class y and actually are of class y; 2 instances are predicted as class y but actually are of class x; 3 instances are predicted as class x but actually are of class y. According to the definition of FP, FP for class x is [(7+3)-7]/(3+4)=0.45. So for class x 45% of instances which are not of class x are predicted as of class x.

Thus it can exhibit values in a range from 0 (excellent model) and 1 (pure classification capabilities of the model). The higher the accuracy, the greater number of instances are classified correctly. However, we cannot rely on accuracy only, since it cannot inform us on the model's quality. The lower the FP is, the more reliable the model is. The FP is computed for each class individually; however for interpretability's sake we are going to present an averaged value provided by WEKA. We use average value in order to present the overall model's FP.

5.5.1 Structured Grids Training Set

In the following subsection we are going to give a detailed explanation of our observations when various predictive models are tested on different versions of training data sets. Each of the items below stand for the cases when we modify the number of instances in a training data set or remove several classes. Every item discusses the highest accuracy observed and the particular training data set preprocessing methodology used (normalization, standardization, attributes selection, etc.). We start exploration of the models with an initial training data set, which is formed using data gathered from the following applications: *myocyte, particle filter, srad, hotspot* . We try models listed in 5.3.2 for each of the version of the training set (5.5).

- *Original training data set (later referred to as OTS).* The maximum accuracy - *22%* and FP is equal to *0.11* - are achieved for a version of the training data set, when numerical values are normalized using WEKA attribute normalization filter, all attributes saved, Random Forest Classifier is used. Generally, for the original data set reduction of the number of attributes does not pay off. Since the data set is not preprocessed at all, it has the lowest number of instances. In that case the model may not have all the knowledge about the relationships inside the data set. That is why each attribute can be valuable. However, FP is considerably low, which indicates that the predictions made are reflecting the relationship among the instances in a certain class;

- *Original training data set with classes s_5, d_5, g_5 (5.4.2.2) removed (later referred to as OTS_2-10).* The maximum accuracy observed is equal to *28%*, FP is equal to *0.152*for normalized training data set with all attributes presented, Random Forest Classifier. As we can see, the accuracy is improved - the number of correctly classified instances are increased. As we mention in 5.5 removing classes _5 enables the model to separate the instances better. However, a lower value for FP requires us to improve the model further on. One important point needs to be mentioned - if after removing the number of classes we move to another model improvement technique, the number of classes is recovered. Next we are trying to run the application with more problem sizes, the number of classes is the original one - 10 classes (classes s_5, d_5, g_5 are recovered back). We perform the process to experiment with various versions of data sets to find out which data set changes are leading to a model's improvement. To provide a reader with a better understanding of the reasons why we remove s_5, d_5, g_5 classes and why that step pays off in terms of accuracy we are going to give the following example. Let's consider the following measurements for execution time (in microseconds) for SRAD application (5.2):

  Measurements for static, dynamic and guided scheduling schemes and a chunk size equal to 2% of a problem size:

Static = 239.6, dynamic = 213.2, guided = 223.8;

Measurements for static, dynamic and guided scheduling schemes and a chunk size equal to 5% of a problem size:

Static = 222.6, dynamic = 205, guided = 196.2;

Measurements for static, dynamic and guided scheduling schemes and a chunk size equal to 10% of a problem size:

Static = 214.8, dynamic = 193.6666667, guided =188.25;

As we can see, the difference between execution time for static policy when chunk sizes are 5% and 10% is only 7.8 microseconds; for dynamic scheduling policy and chunk sizes 2% and 5% the difference is 8.2 microseconds; for guided scheduling policy and chunk sizes 5% and 10% the difference is 7.95 microseconds. When we classify a runtime configuration we find a minimum of execution time achieved when various combination of scheduling policies and chunk sizes are used, thus when chunk size of 5% is considered the categorization is dependent on just a few microseconds. It means that classes which involve chunk size of 5% are very similar to classes involving either chunk size 2% or 10%. It leads to a conclusion that when a machine learning algorithm tries to classify an instance involving chunk size 5% minor fluctuation in values of the attributes are taken into consideration, this situation is called "overfitting". In other words, a model cannot distinguish well between the classes. Thus, removing classes involving chunk size 5% eliminates this problem - classes involving chunk sizes 2% and 10% are more distinguishable and the model can separate the instances more distinctly. Later in this work class removal enables accuracy improvement because of the reasons mentioned in this section;

- *Training data set built with new problem sizes included (later referred to as NewPS).* To enlarge our training set we run applications pertaining to the dwarf with new problem sizes. The best accuracy monitored is *24%* , FP equal to *0.114* , training data set normalized by the total number of execution cycles, all attributes presented, Random Forest Classifier. Accuracy rate and FP lie in the same range as previously observed. That means that adding new information does not improve the model significantly, as well as it does not lower the accuracy. A greater number of observations included in a training data set provides more information on underlying relationships between the attributes and a corresponding class, thus a model learns more precisely. That leads us to a conclusion that we have to consider the new data and try to improve the model's characteristics further on;

- *Training data set built with new problem sizes included and classes s_5, d_5, g_5 removed (later referred to as NewPS_2-10).* The highest value of accuracy observed is *34%* , FP equal to *0.127* initial training data set, all attributes presented, Random Forest Classifier. As we can see, enlarging our training data set and removing certain classes pays off in terms of accuracy. Now one-third of all instances is recognized correctly. FP is slightly increased, meaning that the number of classes does affect a model's quality slightly, however the increase is about 0.09%, thus we can state that class removal generally has no significant effect on model's quality. Hence, taking a higher accuracy rate into account, we can state that class number reduction does pay off ;

- *Training data set built with a new application added to a dwarf being explored (later referred to as NewApp)* . We included Stencil benchmark (5.2) performance to our training data set. The accuracy dropped to the value of *21%* , FP is equal to *0.102* , training data set normalized by the total number of execution cycles, all attributes are saved, Random Forest Classifier. The decrease in the number of instances classified correctly can be explained by the fact that the new application, namely the computations inside the loop that is investigated is not that similar to that of computation from other application. In that case the predictive model cannot firmly infer which attributes values stand for which class. A lower value of FP confirms the stated point. However, we would want to make our model robust, in order to do so we stress in further on;

- *Training data set built with a new application added to a dwarf being explored and classes s_5, d_5, g_5 removed (later referred to as NewApp_2-10).* The accuracy increased up to *31%*, FP is equal to *0.123* . This result is observed for both normalized and standardized versions of a training data set, all attributes preserved, Random Forest Classifier. Removing several classes improved the model slightly, however, the accuracy is lower than previous results showed, and still, all attributes are involved. A higher value for FP increase slightly (by 17 %), thus the quality of the model is not affected significantly;

- *NewApp_2-10 with resample filter applied (later referred to as NewApp_2-10-resample)* . Since the latest training data set we explored includes the greatest number of instances, we would want to improve the model for this data set. We are interested in classifying as many instances as possible to produce a robust prediction. The best accuracy achieved is equal to *73%* , FP is equal to *0.049* , normalized data set with several attributes selected (compiler, execution time, PAPI_L1_DCM, PAPI_L1_TCM, PAPI_FPU_IDL, PAPI_TLB_IM, PAPI_L1_DCH, PAPI_L1_ICH, PAPI_L1_ICA). The resampling creates a random subsample of the original training set. It randomly takes an instance from the original data set and uses it to form a new subsample. In or work we created a subsample which includes the same number of instances that the original one does. We used resampling with replacement, in this case an instance can from the original data set can appear more than one time in a new training data set. This leads to a situation when instances of larger classes are reproduced more frequently and the resulting training set is biased by larger classes. This explains the fact that more instances are classified correctly and false positive metric is lower. Our experiments show that resampling with no replacement does not have a significant impact on accuracy due to the fact that it produces the same training data set. The resampling technique is able to improve a model's accuracy significantly, FP also decreased, meaning that resampling does improve the model's quality. However, since this approach resamples instances based on classes to which they pertain, a resulting model is biased by classes, which include a greater number of instances. Later in this work the results achieved using resampling are reasoned in the same manner as in this section;

- *NewApp_2-10 with datafly filter applied (later referred to as NewApp_2-10-datafly)* . To improve our model further, we employed this filter. The model's accuracy increased up to *96 %* , FP is equal to *0.007* , training data set normalized by the number of total execution cycles, reduced number of attributes (compiler, problem size, PAPI_L1_DCM, PAPI_L1_ICM, PAPI_L2_ICM, PAPI_TLB_TL, PAPI_STL_ICY, PAPI_TOT_INS, PAPI_BR_INS, PAPI_VEC_INS, PAPI_L1_DCH, PAPI_L2_DCH, PAPI_L2_DCA, PAPI_L1_ICH, PAPI_L2_ICR, PAPI_FML_INS, PAPI_FP_OPS, PAPI_DP_OPS). As we can see, attribute selection does pay off. It can be explained by the fact that not all of the attributes represent relevant information. These irrelevant attributes do not provide enough information to separate the instances into classes. Often, redundant attributes are those whose values do not change much for instances pertaining to different classes. Thus, when irrelevant attributes are removed their minor fluctuations are not considered. We performed attribute selection via evolutionary search. This algorithm explores all possible combinations of attributes using genetic approach. Generations of attributes are created (through mutation and crossover operations) and the best generation is chosen. The best means the attributes selected separate the space of the instances more distinguishably than others. Hence, relevant attributes enable better separation of the instances which leads to a higher accuracy rate. FP rate also benefits from attribute selection because those minor fluctuations of irrelevant attributes do not mislead the model anymore. As we can see from the value of FP, Datafly algorithm improved the model's quality to a considerable degree, making it both accurate and reliable. Other machine learning algorithms showed accuracy results near 80-85%, even though the result is satisfactory, in our experiments random forest algorithm showed a better degree of robustness due to its ability to overcome class the over-fitting problem and outlier identification. Later in this work the results achieved using attribute selection and the Datafly algorithm are reasoned in the same manner as in this section;

For the SG dwarf, we can state the most important thing that allowed us to improve the model is presenting attributes with ranges. The advantage of using ranges instead of exact numerical values is that the model becomes less sensitive to the attributes values. This leads to the fact that outliers in a training model does not skew the model. The fewer attributes are used to separate the training set because since we use ranges, the possibility of overitting reduces (exact values of the attribute do not point to a particular class).

The following plot presents the way a model's accuracy changes with respect to various preprocessing techniques used (5.2).

To summarize (5.2), for a Structured Grid dwarf a Random Forest machine learning algorithm appears to be the one which produces the most accurate predictions as opposed to other algorithms. It shows a certain degree of robustness. Normalization of a training data set can be a good technique to use because it allows unifying instances making it easier to distinguish between samples pertaining to the same class.

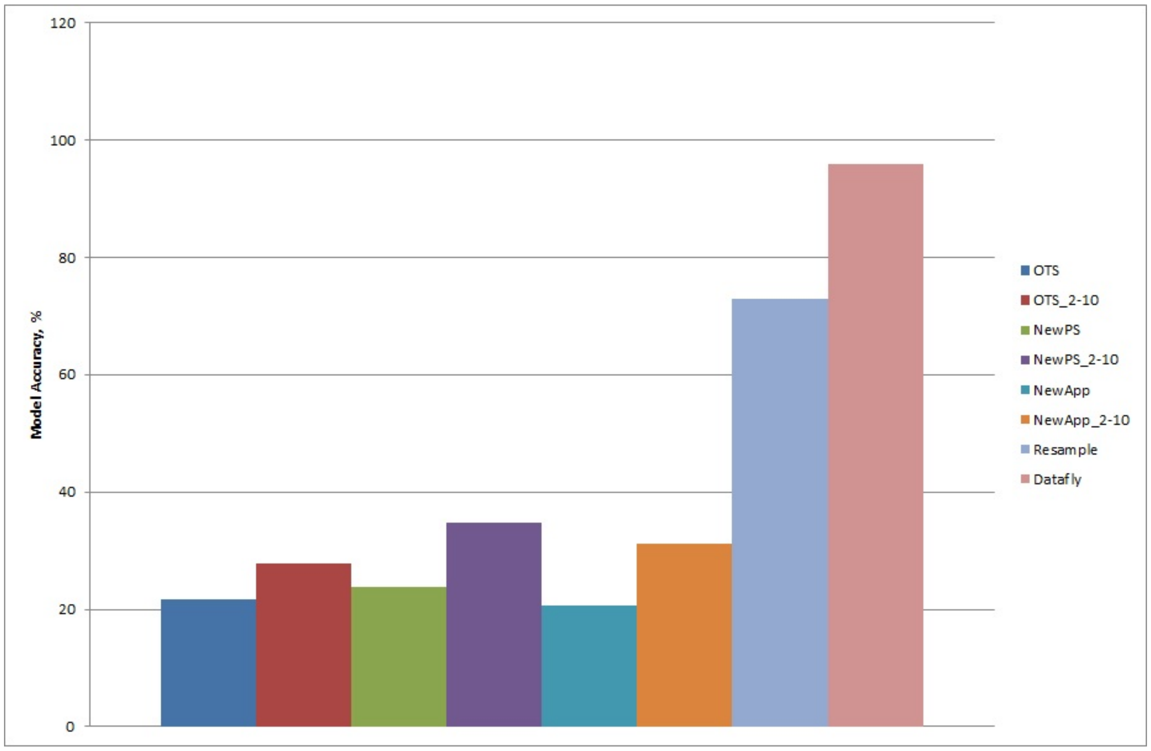


Figure 5.2: Structured Grid dwarf, a model's accuracy vs. preprocessing techniques employed

Reduction in the number of attributes also pays off - less attributes in conjunction with k-anonymity approach facilitates separation of the instances.

5.5.2 Unstructured Grids Training Set

The application which are used to complete the training data set: CFD, back-propagation (5.3.2). The flow of our analysis of the training set follows the same pattern as in 5.5.1.

- *OTS*. The highest value for accuracy is equal to *47%*, FP is equal to *0.104* standardized training data set, all attributes in present, Random Forest Classifier. For an unprocessed data set the result observed seems to be good. This is due to the fact that only two applications are used to complete training data set, hence the possibility that computations inside the loops being tested are drastically different is lower. FP rate is considerably low, thus a majority of instances classified as a particular class are truly of this class;

- *OTS_2-10*. After several class labels are removed, the accuracy goes up to *48%* , FP is equal to *0.154* , unchanged training data set, all attributes in present, Random Forest Classifier. In this case, class removal produces a slight improvement. Since we have fewer instances, even with classes removed, the relationships between samples stay the same. There is a probability that in the initial data set, there are few samples with _5 classes. As for FP rate, the class number reduction increases the rate indicating that more instances classified as a particular class are possibly pertaining to a different one. Since the training data set is not unified in an way, the model could possibly be skewed by outliers;

- *NewPS*. Adding new problem sizes, and thus enlarging the training data set, produces a lower accuracy of *34%* . FP is equal to *0.118*, training data set normalized by the total number of execution cycles, Random Forest Classifier. The decrease in an accuracy rate is due to a greater number of instances in a training data set - now the differences in computation patterns involved are more influential;

- *NewPS_2-10.* The best accuracy observed is *46%* , FP is equal to *0.165* , and the training data set is normalized by the total number of execution cycles, Random Forest classifier. The result is returned to the value observed in the original data set. The reason for this is that the class number reduction enables reduction of an impact of differences between the computations inside the loops. Increased FP rate indicates that the accuracy achieve should be carefully interpreted as more instances are misclassified;

- *NewPS_2-10 with resampling applied.* After random resampling took place, the accuracy of the model increased up to *78%* , with FP is equal to *0.054*, and training data set normalized by the total number of execution cycles. All attributes are preserved, Random Forest Classifier. As mentioned earlier, resampling makes the training data set more random; however, the accuracy of this result cannot be robust enough, since the model is biased by the classes which have a greater number of instances. FP rate is the lowest observed so far; thus, resample does improve model's quality to a certain extent. This improvement is limited by the possibility of bias incurred by classes having greater number of instances;

- *NewPS_2-10 with Datafly applied.* The accuracy received *99%* , FP is equal to *0.004* . This result is observed for a couple of machine learning algorithms: Bayes' Network, Neural Network, Random Forest classifier. However, only the Random Forest algorithm proved to be the most robust one. The version of the training data set that showed the result is a normalized data set with reduced number of attributes (prob_size, PAPI_L1_DCM, PAPI_L2_TCM, PAPI_FPU_IDL, PAPI_BR_MSP, PAPI_FP_INS, PAPI_L1_DCA, PAPI_L2_TCH, PAPI_L2_TCH, PAPI_SP_OPS, PAPI_DP_OPS). The low value of FP indicates that the accuracy achieved reflects the actual relationship among the instances of the training data set.

The following plot presents the way a model's accuracy changes with respect to various preprocessing techniques used:

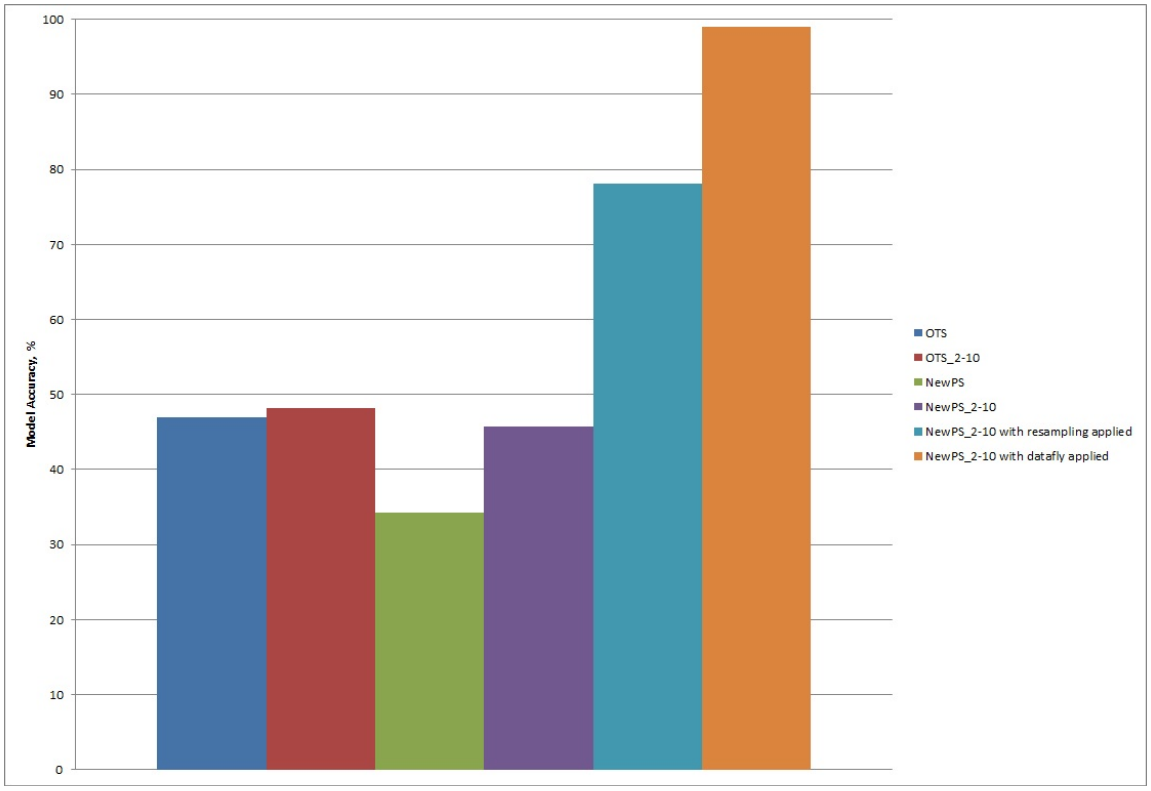


Figure 5.3: Unstructured Grid dwarf, a model's accuracy vs. pre-processing techniques employed

For the current dwarf a Datafly preprocessing filter proves to be a technique, which improves the model's accuracy considerably, as well as instances unification via normalization. The result for this dwarf is higher than for a Structured Grid dwarf because in UG dwarf, we explore two applications; hence, the model may over-fit the values to a certain degree.

5.5.3 Dense Linear Algebra Training Set

Initially, we test the following application to gather performance data: kmeans clustering, streamcluster, LUD solver. The flow of our analysis of the training set follows the same pattern as in 5.5.1.

- *OTS.* The prediction accuracy achieved is *29%* , FP rate is equal to *0.174* , standardized training data set, all attributes present, Random Forest Classifier. The DLA training data set is noticeably larger than previously explored data sets. It leads to the differences in computations in the loops to stand out even more.However, the accuracy of the model is not robust enough since FP rate is considerably high;
- *OTS_2-10.* After removing classes responsible for chunk size equal to 5, the accuracy increases up to *41%*, FP rate is equal to *0.227* , normalized training data set, all attributes present, Random Forest Classifier. We achieve improvement in accuracy; however, FP rate is lower due the fact that some instances which used to belong to removed classes are considerably close to each other (in terms of attributes values); thus, a model tries to fit the instances too precisely;
- *NewPS.* New problem sizes used for taking measurements produced *28%* accuracy, FP rate is equal to *0.114* , a training data set normalized by the number

of total execution cycles, all attributes saved, Random Forest Classifier. As we can see, the value of correctly classified samples is lower than previously observed. As mentioned earlier, new measurements add instances to the training set, hence the relationship between attributes and classes is closer to the real one. In reality, to exhibit high accuracy rates on an unchanged training data set, the computations inside the loops being tested should be nearly the same. A lower FP rate indicates that adding new problem sizes does decrease the true accuracy of the model since the classification function becomes more complex;

- *NewPS_2-10.* The highest accuracy value is equal to *36%* , FP rate is equal to *0.144* , unchanged data set, all attributes present, Random Forest Classifier. As observed earlier, class number reduction does pay off, however, a larger data set still shows that the instances are not of a unified type and behavior. The last point is also confirmed by the FP rate value;

- *NewApp.* We added Matrix Multiplication (MM) application performance measurements to our training data set. The accuracy achieved is *27%* , FP rate is equal to *0.123* , normalized training data set, all attributes are saved, Random Forest Classifier. As previously, adding new instances to the training data set lowers accuracy. However, the fact that the result is near (both accuracy and FP rate) to the one for NewPS signifies that the model is stable to a certain extent;

- *NewApp_2-10.* Removing _5 classes increases the percentage of correctly classified instances o *34%* , FP rate is equal to *0.168* , normalized training data set, all attributes present, Random Forest Classifier. From the previous point we can infer that the model behaves as we expected. The accuracy and FP rate are almost equal to that of NewPS_2-10. It allows us to confirm that the model is stable to a certain degree;

- *NewApp_2-10-resample*. The accuracy rate produced is *73%*, FP rate is equal to *0.077* , normalized training data set, several attributes selected: execution time, PAPI_L1_DCM, PAPI_FPU_IDL, PAPI_STL_ICY, PAPI_BR_TKN, PAPI_BR_MSP, PAPI_FP_INS, PAPI_VEC_INS, PAPI_TOT_CYC, PAPI_L2_DCH, PAPI_L1_DCA, PAPI_L2_ICH, PAPI_L1_TCH, PAPI_FP_OPS, Random Forest Classifier. Resample technique shows improvement in prediction ability and the quality of the classifier (low value for FP rate) for the model. However, for the reasons indicated earlier in this chapter we would to improve the model further on;

- *NewApp_2-10-datafly*. The highest value for accuracy rate is equal to *94 %* , FP rate is equal to *0.016* , training data set normalized by the number of execution cycles, several attributes selected (prob_size, execution time, PAPI_L1_DCM, PAPI_L2_ICM, PAPI_L1_TCM, PAPI_L2_TCM, PAPI_FP_INS, PAPI_BR_INS, PAPI_L2_DCA, PAPI_L2_ICH, PAPI_L2_ICA, PAPI_L2_TCH, PAPI_L1_TCA, PAPI_FAD_INS, PAPI_FSQ_INS, PAPI_DP_OPS) Random Forest Classifier. As observed earlier, k-anonymity approach does improve prediction rates (both accuracy and FP rate) significantly.

  The following picture summarizes all the steps we took while improving prediction produced by the model:

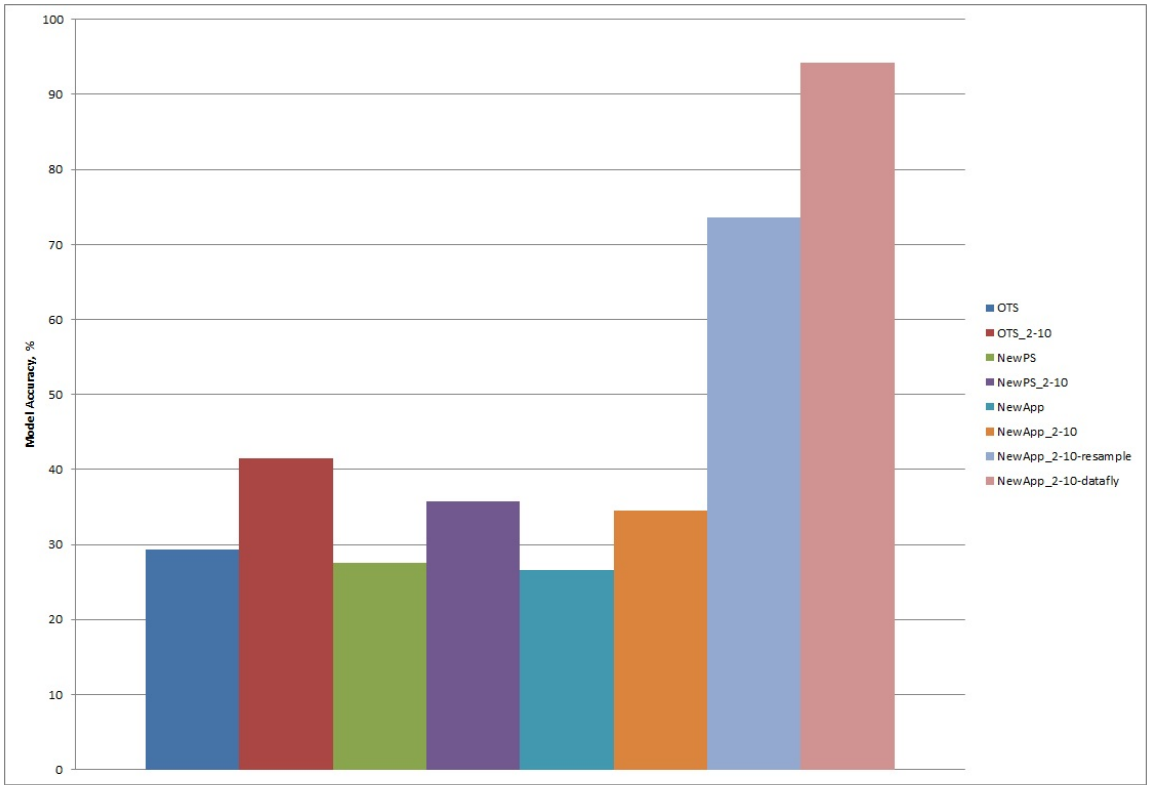


Figure 5.4: Dense Linear Algebra dwarf, a model's accuracy vs. preprocessing techniques employed

To summarize (5.4), for DLA dwarf Datafly, a preprocessing filter allows us to improve the model significantly.

5.6 OpenMP CollectorAPI, TAU Profiling Toolkit

On the final stage of our work we explore several applications from different dwarfs using Collector API and TAU profiling tool (3.9).

To enable Collector API instrumentation of the code we compile the application using TAU's *tau_cc.sh* compiler script. As a result, a number of profiles are created.

Each of those files lists various performance information on a per-thread basis (parallel regions a thread encountered, implicit barriers seen, and exclusive/inclusive time spent in a function/parallel region).

Our analysis is summarized in table 5.6. In particular, we investigate the percentage of a thread's total time spent in a parallel region related to the loop being tested (the problem size is indicated in brackets).

Table 5.6: Per-thread exclusive execution time in a parallel region, %

	Thread 0	Thread 1
DLA		
MM-(64)	6	98
MM-(128)	44.9	99.8
MM-(256)	71.6	99.9
LUD-(64)	0.8	14.3
LUD-(128)	5.6	29.3
LUD-256	11.9	41.3
SG		
Hotspot-(64)	1.2	56.9
Hotspot-(128)	3.30	76.2
Hotspot-256	5.3	78.2
Particle filter-(64)	10.5	57.3
Particle filter-(128)	20.1	59.2
Particle filter-(256)	25.6	58.4

As we can tell from the table 5.6, for the DLA dwarf increasing the problem size leads to a better per-thread workload balancing. For the SG dwarf we can state that the time spent by Thread 1 does not change significantly, as opposed to the execution time spent by Thread 0.

The behavior described is common to applications pertaining to the same dwarf. That leads us to a conclusion that the loops we investigate in different applications do have similarities which allowed us to build a model with a considerably high accuracy of prediction.

In this chapter, we have analyzed our experiments and presented findings. We discussed techniques used to gather training data for further model construction and validation.

The following chapter summarizes the analysis we performed.

Chapter 6

Conclusion and Future Work

The following chapter concludes the work we have done and proposes ways to investigate the subject further on.

6.1 Conclusion

We have examined applications from three different dwarfs - Structured Grid, Unstructured Grid, and Dense Linear Algebra. Our work is aimed at construction of a model in order to predict a combination of scheduling policy and a chunk size, which provides a developer with the best performance on a certain platform for a particular runtime configuration.

Our observations show that unprocessed training data sets tend not show high values of prediction accuracy. The reason for this is the fact that even though the computation in the loops explored pertain to the same computation pattern, the calculations differ from each other in the number of operations, memory access pattern, etc. Benchmarks' vendor claimed the applications we use pertain to a certain dwarf; however, the computations do deal with a particular type of data structures, but the very computation pattern is not the same in all the loops.

The following concepts enable us to improve the models' prediction accuracy significantly:

- *Attributes selection.* This technique enables the model's feature selection and leaves only the attributes which separate the space of samples more obviously. This can be explained by the fact that the exact values of the attributes are used. It leads to a considerable number of cases when a model needs to be fitted too precisely. Finally, attribute selection eliminates redundant information which often leads to separation of instances into different classes even though they should pertain to the same class;

- *Enlarging a training data set.* By making a training set larger, we are able to observe the relationship between instances of different classes with a higher proximity to real life cases. The quality of the prediction is also affected in a good way; since, with more training instances a classification function can be learned more efficiently;

- *Random resampling of the instances.* Resampling a training data set randomizes the data set; thus, making instances less sensitive to their neighbors. This technique improves the prediction accuracy rate significantly, however, due to the fact that there is a possibility of prediction bias in favor of classes which have a greater number of instances, we do not consider this approach as our final solution;

- *K-anonymity data preprocessing.* This technique enables the model to learn on the training data set based on the ranges of the numerical attributes; for each class a set of relevant attributes is defined. This enables the model to classify instances pertaining to the same class based not on exact values of the attributes, but based on the ranges, providing a more general training data set exploration with a less degree of overfitting. Also, irrelevant attributes are recognized on a per-class basis. The quality of the model has also been significantly improved.

Finally, we find out that for each of the dwarfs explored a k-anonymity approach enables the model to improve its characteristics and benefit from various steps taken (data set extension, number of classes reduction, attribute unification). Random Forest Algorithm shows the highest accuracy and FP rates due to its ability to work with complex classification problems.

6.2 Future Work

To improve our models prediction accuracy rates and increase the degree of the model's robustness, we would want to take the following steps:

- Investigation of different architectures;

- Explore new applications which can be related to the dwarfs discussed;
- Integration of a model into a compiler's infrastructure.

Bibliography

[1] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, *et al.*, "The landscape of parallel computing research: A view from Berkeley," tech. rep., Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[2] G. Marin and J. Mellor-Crummey, "Cross-architecture performance predictions for scientific applications using parameterized models," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 32, pp. 2–13, ACM, 2004.

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

[4] X. Wu and V. Taylor, "Performance modeling of hybrid mpi/openmp scientific applications on large-scale multicore cluster systems," in *Computational Science and Engineering (CSE), 2011 IEEE 14th International Conference on*, pp. 181–190, IEEE, 2011.

[5] C. Dubach, T. Jones, and M. O'Boyle, "Exploring and predicting the architecture/optimising compiler co-design space," in *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pp. 31–40, ACM, 2008.

[6] J. Cavazos, C. Dubach, F. Agakov, E. Bonilla, M. O'Boyle, G. Fursin, and O. Temam, "Automatic performance model construction for the fast software exploration of new hardware designs," in *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pp. 24–34, ACM, 2006.

[7] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. O'Boyle, and O. Temam, "Rapidly selecting good compiler optimizations using performance counters,"

in *Code Generation and Optimization, 2007. CGO'07. International Symposium on*, pp. 185–197, IEEE, 2007.

[8] *OpenMP programming model.* `http://en.wikipedia.org/wiki/OpenMP`.

[9] K. Yue and D. Lilja, "Parallel loop scheduling for high performance computers," *Advances in Parallel Computing*, vol. 10, pp. 243–264, 1995.

[10] C. Bishop *et al.*, *Pattern recognition and machine learning*, vol. 4. springer New York, 2006.

[11] C. Liao, O. Hernandez, B. Chapman, W. Chen, and W. Zheng, "Openuh: An optimizing, portable openmp compiler," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 18, pp. 2317–2332, 2007.

[12] *The GNU Compiler.* `http://gcc.gnu.org/onlinedocs/gcc/`.

[13] *The Portland Group Compiler.* `http://www.mcsr.olemiss.edu/bookshelf/doc/man/man1/pgcc.htm`.

[14] *PAPI.* `http://icl.cs.utk.edu/papi/index.html`.

[15] O. Hernandez, R. C. Nanjegowda, B. Chapman, V. Bui, and R. Kufrin, "Open source software support for the openmp runtime api for profiling," in *Parallel Processing Workshops, 2009. ICPPW'09. International Conference on*, pp. 130–137, IEEE, 2009.

[16] *TAU profiling toolkit.* `http://tau.uoregon.edu`.

[17] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pp. 44–54, IEEE, 2009.

[18] *SHARK.* `http://pstl.cs.uh.edu/resources/shark`.

[19] *Parboil Benchmark Suite.* `http://impact.crhc.illinois.edu/parboil.aspx`.

[20] *WEKA data mining and machine learning software.* `http://www.cs.waikato.ac.nz/ml/weka/`.

[21] *BN example.* `https://controls.engin.umich.edu/wiki/index.php/Bayesian_network_theory#Worked_out_Example_2`.

[22] E. Frank, M. Hall, L. Trigg, G. Holmes, and I. Witten, "Data mining in bioinformatics using weka," *Bioinformatics*, vol. 20, no. 15, pp. 2479–2481, 2004.

[23] C. Lively, X. Wu, V. Taylor, S. Moore, H. Chang, C. Su, and K. Cameron, "Power-aware predictive models of hybrid (mpi/openmp) scientific applications on multicore systems," *Computer Science-Research and Development*, pp. 1–9, 2011.

[24] K. Huck, O. Hernandez, V. Bui, S. Chandrasekaran, B. Chapman, A. Malony, L. McInnes, and B. Norris, "Capturing performance knowledge for automated analysis," in *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pp. 1–10, IEEE, 2008.

[25] M. Voss, K. Yau, R. Eigenmann, *et al.*, "Interactive instrumentation and tuning of openmp programs," *Proc. of PDPTA 2000 (Parallel and Distributed Processing Techniques and Applications), Las Vegas, NV, USA*, 2000.

[26] A. Duran, J. Corbalán, and E. Ayguadé, "Evaluation of openmp task scheduling strategies," *OpenMP in a New Era of Parallelism*, pp. 100–110, 2008.

[27] A. Baddeley and R. Turner, "Spatstat: an r package for analyzing spatial point patterns," *Journal of Statistical Software*, vol. 12, no. 6, pp. 1–42, 2005.

[28] Y. Bengio and Y. Grandvalet, "No unbiased estimator of the variance of k-fold cross-validation," *The Journal of Machine Learning Research*, vol. 5, pp. 1089–1105, 2004.

[29] L. Breiman, "Statistical modeling: The two cultures (with comments and a rejoinder by the author)," *Statistical Science*, vol. 16, no. 3, pp. 199–231, 2001.

[30] B. Wicaksono, R. C. Nanjegowda, and B. Chapman, "A dynamic optimization framework for openmp," in *OpenMP in the Petascale Era*, pp. 54–68, Springer, 2011.

Appendix A

Hardware Counters Available on Shark Platform

In this part of our work we present a list of hardware countres available on the platform explored.

PAPI_L1_DCM - Level 1 data cache misses

PAPI_L1_ICM - Level 1 instruction cache misses

PAPI_L2_DCM - Level 2 data cache misses

PAPI_L2_ICM - Level 2 instruction cache misses

PAPI_L1_TCM - Level 1 cache misses

PAPI_L2_TCM - Level 2 cache misses

PAPI_FPU_IDL - Cycles floating point units are idle

PAPI_TLB_DM - Data translation lookaside buffer misses

PAPI_TLB_IM - Instruction translation lookaside buffer misses

PAPI_TLB_TL - Total translation lookaside buffer misses

PAPI_STL_ICY - Cycles with no instruction issue

PAPI_HW_INT - Hardware interrupts

PAPI_BR_TKN - Conditional branch instructions taken

PAPI_BR_MSP - Conditional branch instructions mispredicted

PAPI_TOT_INS - Instructions completed

PAPI_FP_INS - Floating point instructions

PAPI_BR_INS - Branch instructions

PAPI_VEC_INS - Vector/SIMD instructions (could include integer)

PAPI_RES_STL - Cycles stalled on any resource

PAPI_TOT_CYC - Total cycles

PAPI_L1_DCH - Level 1 data cache hits

PAPI_L2_DCH - Level 2 data cache hits

PAPI_L1_DCA - Level 1 data cache accesses

PAPI_L2_DCA - Level 2 data cache accesses

PAPI_L1_ICH - Level 1 instruction cache hits

PAPI_L2_ICH - Level 2 instruction cache hits

PAPI_L1_ICA - Level 1 instruction cache accesses

PAPI_L2_ICA - Level 2 instruction cache accesses

PAPI_L1_ICR - Level 1 instruction cache reads

PAPI_L1_TCH - Level 1 total cache hits

PAPI_L2_TCH - Level 2 total cache hits

PAPI_L1_TCA - Level 1 total cache accesses

PAPI_L2_TCA - Level 2 total cache accesses

PAPI_FML_INS - Floating point multiply instructions

PAPI_FAD_INS - Floating point add instructions

PAPI_FDV_INS - Floating point divide instructions

PAPI_FSQ_INS - Floating point square root instructions

PAPI_FP_OPS - Floating point operations (Counts speculative adds and multiplies. Variable and higher than theoretical.)

PAPI_SP_OPS - Floating point operations; optimized to count scaled single precision vector operations

PAPI_DP_OPS - Floating point operations; optimized to count scaled double precision vector operations

Appendix B

Framework for Taking Measurements

In this part of our work we are going to present three parts of framework developed for taking measurements.

Content of *init.h*

```
int retval;
long long values[3], start_usec, end_usec;
int *group1;
int EventSet = PAPI_NULL;
int private_event_set=PAPI_NULL;
int loop_counter;
int nthreads=0;
```

```
char *cvalue = NULL;
int gr_id ,NUM_EVENTS, problem_size ,mode;
float chunk_size;
int *final_values;

static const char filename[] = "counters_data.txt"; //time_data.txt
FILE *file = fopen ( filename, "r" );
char line[256];
int linenum=0;

while(fgets(line, 256, file) != NULL)
{
                char gr[256], m[256];
if(linenum==file_line)
{
if(sscanf(line, "%s %s", gr, m) != 2)
{
fprintf(stderr, "Syntax error, line %d\n", linenum);
continue;
}
gr_id=atoi(gr);
mode=atoi(m);
chunk_size=(float)gr_id/100;
}linenum++;
}
if ( ( retval = PAPI_library_init( PAPI_VER_CURRENT ) ) != \
        PAPI_VER_CURRENT )
fprintf(stderr,"PAPI␣library␣version␣mismatch!\n");
switch (gr_id)
{
                case 0:

                NUM_EVENTS=3;
                break;

                case 1:

                NUM_EVENTS=2;
                break;

                case 2:

                NUM_EVENTS=3;
                break;
```

```
case 3:

NUM_EVENTS=2;
break;

case 4:

NUM_EVENTS=3;
break;

case 5:

NUM_EVENTS=3;
break;

case 6:

NUM_EVENTS=3;
break;

case 7:

NUM_EVENTS=1;
```

```
break;

case 8:

NUM_EVENTS=3;
break;

case 9:

NUM_EVENTS=1;
break;

case 10:

NUM_EVENTS=3;
break;

case 11:

NUM_EVENTS=2;
break;

case 12:
```

```
                    NUM_EVENTS=3;
                    break;

                    case 13:

                    NUM_EVENTS=3;
                    break;

                    case 14:

                    NUM_EVENTS=2;
                    break;

                    case 15:

                    NUM_EVENTS=3;
                    break;

                    }
group1 = (int *) malloc (NUM_EVENTS * sizeof(int));
final_values = (int *) malloc( NUM_EVENTS* sizeof(int));
int mm;
```

```
for (mm=0;mm<3;mm++)
final_values [mm]=0;
switch (gr_id)
{
case 0:
group1[0]=PAPI_L1_DCM;
group1[1]=PAPI_L1_ICM;
group1[2]=PAPI_L2_DCM;
NUM_EVENTS=3;
break;

case 1:
group1[0]=PAPI_L2_ICM;
group1[1]=PAPI_L1_TCM;
NUM_EVENTS=2;
break;

case 2:
group1[0]=PAPI_L2_TCM;
group1[1]=PAPI_FPU_IDL;
group1[2]=PAPI_TLB_DM;
NUM_EVENTS=3;
break;
```

```
case 3:
group1[0]=PAPI_TLB_IM;
group1[1]=PAPI_TLB_TL;
NUM_EVENTS=2;
break;

case 4:
group1[0]=PAPI_STL_ICY;
group1[1]=PAPI_HW_INT;
group1[2]=PAPI_BR_TKN;
NUM_EVENTS=3;
break;

case 5:
group1[0]=PAPI_BR_MSP;
group1[1]=PAPI_TOT_INS;
group1[2]=PAPI_FP_INS;
NUM_EVENTS=3;
break;

case 6:
group1[0]=PAPI_BR_INS;
```

```
group1[1]=PAPI_VEC_INS;
group1[2]=PAPI_RES_STL;
NUM_EVENTS=3;
break;

case 7:
group1[0]=PAPI_TOT_CYC;
NUM_EVENTS=1;
break;

case 8:
group1[0]=PAPI_L1_DCH;
group1[1]=PAPI_L2_DCH;
group1[2]=PAPI_L1_DCA;
NUM_EVENTS=3;
break;

case 9:
group1[0]=PAPI_L2_DCA;
NUM_EVENTS=1;
break;

case 10:
```

```
group1[0]=PAPI_L1_ICH;
group1[1]=PAPI_L2_ICH;
group1[2]=PAPI_L1_ICA;
NUM_EVENTS=3;
break;

case 11:
group1[0]=PAPI_L2_ICA;
group1[1]=PAPI_L1_ICR;
NUM_EVENTS=2;
break;

case 12:
group1[0]=PAPI_L1_TCH;
group1[1]=PAPI_L2_TCH;
group1[2]=PAPI_L1_TCA;
NUM_EVENTS=3;
break;

case 13:
group1[0]=PAPI_L2_TCA;
group1[1]=PAPI_FML_INS;
group1[2]=PAPI_FAD_INS;
```

```
NUM_EVENTS=3;
break;

case 14:
group1[0]=PAPI_FDV_INS;
group1[1]=PAPI_FSQ_INS;
NUM_EVENTS=2;
break;

case 15:

group1[0]=PAPI_FP_OPS;
group1[1]=PAPI_SP_OPS;
group1[2]=PAPI_DP_OPS;
NUM_EVENTS=3;
break;

}
if ( ( retval = PAPI_thread_init((unsigned long ( * )\
( void ) )omp_get_thread_num)) != PAPI_OK )
fprintf(stderr,"PAPI thread initialization error\n");
```

Content of *start.h* .

```
if(mode==0)
{
start_usec = PAPI_get_real_usec();
}

# pragma omp parallel shared ( final_values , size,i,a)\
          private ( nthreads,retval,values,loop_counter,j,k,sum) \
          firstprivate(group1,private_event_set,NUM_EVENTS) \

{
nthreads=omp_get_thread_num();

if(mode ==1)
{
if ( ( retval = PAPI_create_eventset( &private_event_set) ) \

fprintf(stderr, 'tid_%d_PAPI_create_eventset!\n',nthreads);

if ( ( retval =  PAPI_add_events( private_event_set, \
                              ( int * ) group1,NUM_EVENTS) ) < PAPI_OK )
fprintf(stderr,"tid_%d_PAPI_add_events!\n",nthreads);
if ( ( retval = PAPI_start( private_event_set ) ) != PAPI_OK )
fprintf(stderr,"tid %d PAPI_start!\n",nthreads);

}
```

Content of *end.h* .

```
if(mode==1)
{
if ( ( retval = PAPI_stop( private_event_set , values ) ) != PAPI_OK )
fprintf(stderr,"PAPI_stop!\n");
for(loop_counter=0;loop_counter<NUM_EVENTS;loop_counter++)
{
#pragma omp atomic
final_values[loop_counter]+=values[loop_counter];
}
}
}//of omp parallel
if(mode==0)
{
end_usec = PAPI_get_real_usec();
printf("%lld",end_usec-start_usec);
}

//}

if(mode==1)
for(loop_counter=0;loop_counter<NUM_EVENTS;loop_counter++)
{
```

```
final_values[loop_counter]=final_values[loop_counter]/omp_get_max_thread
printf("%d,", final_values[loop_counter]);
}

exit(EXIT_FAILURE);
```