THE UNIVERSITY *of* EDINBURGH

Towards Alleviating The Software Parallelization Task

Aleksandr Maramzin

Master of Philosophy

Institute of Computing Systems Architecture

School of Informatics

University of Edinburgh

2021

Abstract

Despite decades of research into parallelizing compiler technology, *software parallelization* remains a largely manual task, which is complex, time-consuming, and error-prone. An embarrassingly parallel problem can be hidden behind a serial algorithm, thoughtless software design, or unsuccessfully chosen lower-level constructs, such as data structures. To elegantly and effectively map a parallel problem onto the exact hardware a programmer must possess expert-level knowledge in various fields from software design and algorithmic patterns down to automatic vectorization and cache coherence. In this thesis, we do not strive to find a ”silver bullet” and solve the problem of automatic parallelization. Neither do we expect an average programmer to be an expert. Instead, we acknowledge the role a programmer plays in the parallelization process and equip the former with an *assistant solution*. Our solution alleviates the task and makes parallelism more accessible to an average programmer.

The assistant solution consists of a tool and a library aiming at different stages of software parallelization. The tool aims at finer granularity levels. Program loops are often the richest source of parallelism and account for the biggest portion of the running time. The tool identifies those loops, which are both worthwhile and feasible to parallelize. For each loop, the tool combines its potential contribution to speedup and an estimated probability for its successful parallelization. This probability is predicted using a machine learning model, which has been trained and tested on 1415 labelled loops, achieving a prediction accuracy greater than 90%. We present a methodology that makes better use of expert time by guiding them directly towards those loops, where the largest performance gains can be expected while keeping analysis and transformation effort at a minimum. We have evaluated our parallelization assistant against sequential C applications from the SNU NAS benchmark suite. We show that our novel methodology achieves parallel performance levels comparable to those from expert programmers while requiring less expert time. On average, our assistant reduces the number of lines of code that have to be inspected manually before reaching expert-level parallel speedup by 20%.

The library implements the novel idea of *computational frameworks*, which are higher-level entities that embody both data structures and algorithms. The use of computational frameworks as parallel software design primitives alleviates the process of parallel software development for a wide class of applications. We prototyped the library on the Olden benchmark suite. The parallel library version consistently outperforms the sequential version hitting 5-6x speedups on the major benchmarks.

Acknowledgements

This work was supported by grant EP/L01503X/1 for the University of Edinburgh School of Informatics Centre for Doctoral Training in Pervasive Parallelism. I would like to express my special gratitude to the UKRI and the CDT in Pervasive Parallelism for the great opportunity they gave me.

The work would not have been done without the constant direction and guidance of my primary supervisor Björn Franke. I would also like to separately thank Murray Cole for his wise advice and support. I express my gratitude to my co-supervisors Michael O'Boyle and Kenneth Heafield for their help in reviewing my project and directing my work.

I would also like to thank all my friends and colleagues for the numerous discussions, technical help, and their time. Special thanks to Artemy, Chris, Nikolay, Roberto, Rodrigo and many, many others. And, of course, I would like to thank my mother for her understanding, encouragement, and support.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Aleksandr Maramzin*)

Contents

Chapter 1

Introduction

1.1 Overview

Parallelism has become pervasive in the world of computing, with parallel hardware omnipresent across the whole spectrum of various computing systems from low-end embedded devices to high-end supercomputers. Yet, most of the existing software is sequential: be it an old legacy software initially designed for the serial hardware or modern applications developed by application domain experts rather than performance engineers. To exploit all available hardware facilities software has to be parallelized.

The software parallelization problem

Manual parallelization challenges The task of software parallelization has characteristically been a very manual process, which is multifaceted, extremely complex, time-consuming, and error-prone. An embarrassingly parallel problem might end up being hidden behind a thoughtless software design, a serial algorithm, or being implemented with unsuccessfully chosen lower-level constructs, such as pointers, heap-allocated and pointer-linked data structures, indirect array referencing, etc. To elegantly and effectively map a parallel problem onto the exact hardware a programmer must work on the various abstract levels and possess expert-level knowledge in various fields from software design and algorithmic patterns in software engineering down to compiler automatic vectorization and hardware cache coherence protocols. It is not always realistic to expect such wide expertise from an average programmer. Done in the wrong way software parallelization can even slow the program down in comparison to its original sequential version.

Automatic parallelization limitations Given the difficulty of manual software parallelization, there have been numerous efforts aimed at automating the task [12]. For several decades, parallelizing compilers have been the subject of intensive academic research [55] and industrial investment (Intel Compiler [3]). Yet, for most real-world applications they still fail to deliver parallel performance, and to fully exploit the potential of modern parallel hardware one still needs to apply a significant manual effort. Furthermore, automatic parallelization techniques are limited to narrow domains of straightforward scientific C and Fortran codes and relatively simple computational idioms. When dealing with arbitrary real-world codes automatically parallelizing compilers face a number of problems and challenges. The *Data-Centric Parallelization (DCP) problem* is an important demonstrative example. Listings 1.1 and 1.2 illustrate the problem and show how easily automatic parallelization[1] can be hampered. In an array-based implementation (Listing 1.1) a compiler knows addresses of all sequence elements statically and can generate the code processing different array elements in parallel, while in a linked-list based implementation (Listing 1.2) we see a pointer chasing code, where addresses of sequence elements will be resolved only dynamically and a compiler cannot generate parallel code in advance. In a real world code, the problem is obviously a way more challenging: *data structures are closely entangled with algorithms*. Parallelization of these codes often requires a human mind.

```
int a[1024];
for (int i=0; i<1024; i++) {
  a[i]=a[i]+1;
}
```

Listing 1.1: Parallelizable loop operating on what is clear to compiler a **linear array**.

```
struct Node* nptr;
for (p=nptr; p!=NULL; p=p->next) {
  p->value+=1;
}
```

Listing 1.2: Non-parallelizable loop operating on what programmer knows is a **linked-list**.

Machine learning based parallelization applicability There were attempts to tackle the problem in another way. There is a vast body of research into utilizing more exotic machine learning based methods in the field of software parallelization. A good overview is provided by [77]. These methods have proved to be extremely useful and high performing on some compilation technology problems like selecting the best compiler flags or finding the most optimal compiler optimization parameters (like loop

[1] By parallelizable loop in this context we mean that auto parallelizing compiler can statically prove the non-existence of loop-carried dependencies and generate parallel or vector code.

unroll or function inline factors). However, due to the inherent statistical errors and unavailability of large training data sets for compilation problems these methods have not yet found a widespread application in the area of software parallelization.

The assistant solution

In this thesis, we are not trying to find a ”silver bullet” and solve the problem of automatic parallelization. Neither do we try to tune machine learning algorithms to a perfect 100% prediction accuracy. Given the difficulty of the obstacles faced by the field today, we do not expect that programmers will be liberated from performing manual parallelization in the near future [53]. Instead, we acknowledge the role of a human programmer in the software parallelization process, but we do not expect the programmer to be an expert either. What we try to do is to ***reduce the manual effort*** involved in the task by providing a programmer with a parallelization *assistant solution*. Our solution alleviates the task and makes parallelism more accessible to an average programmer. The assistant solution we propose is as multifaceted as the problem itself. To fully exploit all the potential of software parallelization a programmer has to work on several conceptual levels. Thus, the assistant solution consists of a machine learning based loop parallelization tool [79] aiming at the finest levels of granularity[2], namely the program loops and a library of computational frameworks [82] aiming at a coarse-grained parallelization on a higher level of software architecture design, algorithm and data structure choice.

1.2 Loop Parallelization Assistant

Despite decades of intensive research in automatic software parallelization [55], fully exploiting the potential of modern parallel hardware still requires a significant manual effort. Chapter 4 introduces a novel parallelization assistant that aids a programmer in the software parallelization process in the frequent case where automatic

[2]In this thesis, by the granularity of parallelizing source code transformation, we mean the scope and complexity of the change. For example, doing array privatization in a loop and adding one OpenMP pragma before it would be a relatively fine-grained parallelizing transformation. This transformation is tiny, does not require the change of algorithm or underlying data structure type. Whereas, identifying a parallel tree reduction in a complex sequential pointer-based code and substituting it with a parallelizable alternative would classify as a coarse-grained parallelization. That transformation would require a lot of effort in identifying the data structure type and proving it is a tree and then substituting the code with a parallelizable alternative. Note, that this terminology might be misleading as it does not always correctly reflect the ultimate parallel speedup: we might have a small, but long-running loop in the code. But most of the time in practice coarse-grained transformations will indeed materialize into more significant performance improvements.

approaches fail. The assistant works at the finer levels of granularity, namely the program loops. Loops are compelling candidates for parallelization, as they are naturally decomposable and tend to capture most of the execution time in a program. The assistant reduces the manual effort in this process by presenting a programmer with a ranking of program loops that are most likely to 1) require little or no effort for successful parallelization and 2) improve the program's performance when parallelized. Thus, it improves over the traditional, profile-guided process by also taking into account the *probability* of potential parallelization for each of the profiled loops.

At the core of our parallelization assistant resides a novel machine-learning (ML) model of loop parallelizability. Focusing on loops allows the model to leverage a large number of specific analyses available in modern compilers, such as generalized iterator recognition [75] and loop dependence analysis [69]. The model encodes the results of these analyses together with basic properties of the loops as machine learning *features*. The loop parallelizability model is trained, validated, and tested on 1415 loops from the SNU NAS Parallel Benchmarks (SNU NPB) [50]. The loops are labelled using a combination of expert OpenMP [22] annotations and optimization reports from the Intel Compiler (ICC), a production-quality parallelizing compiler. The model is evaluated on multiple machine learning algorithms. The evaluation shows that – despite the limited size of the data set – our model achieves a prediction accuracy higher than 90%.

The parallelization assistant combines inference on the parallelizability model with traditional profiling to rank higher those loops with a high probability of being parallelizable and impacting the program performance. An evaluation on eight programs from the SNU NPB suite shows that the program performance tends to improve faster as loops are parallelized in the ranking order suggested by our parallelization assistant compared to a traditional order based on profiling only. On average, following the order suggested by the assistant reduces by approximately 20% the number of lines of code a programmer has to examine manually to parallelize SNU NPB to its expert-level speedup. Given the high level of effort involved in manual analysis, such a reduction can translate into substantial development cost savings.

1.2.1 Contributions of our Loop Parallelization Assistant

In summary, our machine learning based loop parallelization assistant makes the following contributions:

▷ We introduce a machine learning model, which can be used to predict the probability with which sequential C loops can be parallelized (Sections 4.2 and 4.3);

▷ We integrate profiling of execution time with our novel ML model into a parallelization assistant, which guides the user through a ranked list of loops for parallelization (Section 4.4); and

▷ We demonstrate that our tool and methodology increase programmer productivity by identifying parallel loop candidates better than existing state-of-the-art approaches (Section 4.5).

1.3 Computational Frameworks library

If we have a sequential program and want to improve its performance on a parallel machine, we might use our software parallelization assistant to guide a programmer through a list of ranked application loops and advice on where to concentrate manual efforts. The ultimate software transformation is in the hands of a programmer. A programmer might decide to skip the highlighted loop or parallelize it. The loop might already be parallelizable and one OpenMP pragma would be all that is required, or the loop might require some prior enabling transformation before its parallelization can be accomplished. These source code changes would be classified as relatively fine-grained parallelizing transformations but in some cases, to get the best possible performance results, one might want to make more radical changes, i.e change the algorithm, the data structure, or redesign the software architecture starting from the proposed loop. These tasks often require deep expertise from a programmer and are relatively difficult. A programmer would also face the same challenges if he was to design and develop parallel software from the scratch.

To assist a programmer in tackling the problem of parallel software development on a higher level we propose the concept of *computational frameworks* and implement it as a prototype library (see Section 5.3.5). The concept blends algorithms and data structures to form an elegant higher-level entity that could be used as a parallelization primitive. The concept of computational frameworks has been inspired by the problems in the software parallelization field, a relevant concept of algorithmic skeletons (see the comparative analysis of the two concepts in Section 2.7.1), and by the complexities of the real-world legacy code. We have already introduced the data-centric parallelization (DCP) problem. Generally, understanding the data structure type re-

quires a thorough understanding of the algorithm that uses it and vise versa. We call it *the problem of data structure and algorithm inseparability*. For many real-world programs, the task of separating data structures from algorithms is extremely challenging, but for some, it does not seem meaningful either. For example, in many Olden benchmarks (see Section 2.8.3) the data structures and algorithms are blended, but the union they form can be framed into an elegant higher-level entity, that can later be parallelized in a nice and structured way. We provide a programmer with a ready-to-use solution to do that - a library of computational frameworks. If a problem to solve fits into computational patterns our frameworks address, then all a programmer needs to do is to get familiar with library API and write the custom part. The software design and its system-level parallelization have already been taken care of.

We demonstrate the utility and potential of the concept by deploying the library on the subset of the Olden benchmark suite. We express benchmark computations in terms of our computational frameworks and rewrite the original legacy C versions of these benchmarks in a modern, better structured, and crucially parallel way. The parallel library version consistently outperforms the sequential version hitting 5-6x speedups on the major benchmarks.

1.3.1 Contributions of our Computational Frameworks

▷ We propose a novel idea of *computational frameworks*, which are higher-level entities that embody both data structures and algorithms; and

▷ report on a research prototype C++ template library [82] implementing the idea in a modern, convenient, parallel, and easy to use way;

▷ We express computations of some Olden benchmarks in terms of our computational frameworks and rewrite their original sequential legacy C versions with the help of our library in a modern, better structured, portable, and crucially parallel way (see Section 5.3);

▷ We demonstrate the potential of the idea and performance of the prototype library on the suite of Olden benchmarks (see Section 5.4), achieving consistent parallel speedups of 5-6x on the major benchmarks;

▷ Finally, we propose an idea of an alternative software parallelization approach based on our computational frameworks as future work.

Chapter 2

Background and Motivation

Parallel computing and *software parallelization* are vast, overlapping, and complimentary computer science areas with a rich history dating back to the 1950s. With advances in the semiconductor industry, the topics have left the niche of high-end scientific supercomputers and spread to a much wider area spanning across all consumer electronic devices and have become of major importance.

Nowadays, parallelism is pervasive. Parallel hardware is omnipresent across the whole wide spectrum of various computing systems. To exploit all available hardware capabilities software has to be parallel as well. And thus, every computer scientist and software developer would benefit from having an insight into the area. Nonetheless, the topics are extremely complex, require a serious time investment and a great deal of knowledge in various subdomains. It is not realistic to expect an average programmer to possess such deep expertise. For that reason, we propose a solution aimed at alleviating the challenging task of manual software parallelization. Our solution consists of two components. We describe them in Chapters 4 and 5.

In this chapter, we stress the importance of software parallelization, highlight its challenges, describe the parallel software engineering process, and finally lay the ground for our proposed solutions from Chapters 4 and 5. **The major ideas leading towards the solutions from Chapters 4 and 5 are highlighted with boldface text.** We express our special gratitude to Lawrence Livermore National Laboratory (LLNL) [91] for their great parallel computing tutorials. We heavily relied on those to prepare the background material.

The background chapter is structured as follows. Section 2.1 stresses the importance of parallel computing and software parallelization in the modern world. There are numerous software parallelization methods and techniques that address the problem, but

all of them run into specific challenges and limitations. Section 2.2 highlights the major problems of various software parallelization methods. First, it presents the challenges of manual and then automatic software parallelization techniques. There have been various experiments and works applying machine learning (ML) based methods to the field of compilers [77] and software parallelization in particular [58]. The challenges of manual and automatic parallelization combined with the idea of using machine learning based methods lay the ground for our program loop parallelization assistant solution [79]. We describe our assistant solution in Chapter 4.

Section 2.3 discusses the problem of data structure choice and how it affects software parallelization. Unfortunately, there are no universal automatic solutions to this problem at the moment. Data structures are often inseparable from the algorithms they support. Our computational frameworks build on that fact by defining the blend of data structures and algorithms. We propose our solution to the problem in Chapter 5. It is very important to compare the concept of computational frameworks to a well-established concept of parallelism patterns (algorithmic skeletons). Section 2.7.1 presents this comparison. Furthermore, modern software design and engineering tasks are extremely rich and complex topics. Of course, that is true of parallel software engineering as well. In Sections 2.4 and 2.5 we talk about imperative, functional and object-oriented programming paradigms, as well as various OOP software design patterns. In Section 2.7.2 we explain how our computational frameworks take the best from these principles.

2.1 The importance of parallel computing

Parallelism is pervasive and the future of computing is parallel. Many factors stress the importance of parallelism in the modern computing world.

Abundance of natural parallelism The field of High-Performance Computing (HPC) has traditionally been concerned with scientific modeling and simulation of various natural phenomena (climate change, fluid flows, etc.). Physical systems consist of numerous, often independent parts. Moreover, these problems are often expressed through common parallelizable mathematical models: parallel Gauss [43] and Conjugate Gradient methods [48], [52] for solving linear equation systems, parallel matrix inversion [60], parallelizable optimizations for specific cases (like sparse matrices [71], [72]), etc. When we compile a highly parallel algorithm to a serial sequence of CPU in-

structions or process a huge dataset with independent parts sequentially, we artificially constrain a vastly parallel computation to a serial one.

Semiconductor technology advances and power limits With advances in transistor density, it became feasible to design more complicated CPUs. Initially, the trend went towards more complex microarchitecture with deeper pipelines, but running into power limits the industry design shifted towards multi-core CPUs and multiprocessor systems. To exploit such systems fully, software must mirror the trend and become parallel as well.

Domain inherent parallelism and specialized computations The areas like computer graphics for instance have a lot of problems that can be processed in a Single Instruction Multiple Data (SIMD) fashion. That naturally led to the emergence of specialized co-processors like GPUs making hardware systems more complex and heterogeneous [46].

To fully exploit all capabilities provided by modern high-performance computing systems, software has to be mapped onto the parallel hardware i.e parallelized.

2.2 Challenges in software parallelization

The problem of software parallelization is extremely complex and multi-faceted. There are various approaches to the problem, but all of them have their pros and cons. Although the process of software parallelization has characteristically been a very *manual* task, which is time-consuming and error-prone, there are also *automatic* and *machine learning based* techniques. In this section, we highlight the inherent problems of all these approaches. The solution we propose grows on these challenges.

2.2.1 Manual parallelization challenges

Parallel software development has characteristically been a very manual process. Like any software development process, it consists of several stages. The major problems are described below.

Problem understanding and partitioning As the best software engineering practices dictate, before diving into software development one needs to thoroughly understand the problem and decide on the requirements and restrictions the final piece of software

must meet. The whole algorithm and software architecture might change with the decision of developing a parallel software version instead of a serial one. If one starts from an already implemented serial software version, the parallelization might be even more difficult to do. Source code comprehension is a hard task. The algorithm chosen for a serial version might be completely unsuitable for a parallel implementation. The problem must be partitioned into relatively independent chunks of work to be processed in parallel. The partitioning can be done in multiple ways and a programmer needs to choose one (data set decomposition, functional decomposition, or a hybrid of the two).

Communications and synchronization Very often the parts of the problem are not completely independent and require an exchange of information. Designing the way that exchange is going to work is a complex task. Almost always communication results in overhead. Sending the data over a congested network or waiting on a synchronization barrier slows the program down. The slowdown might even diminish all performance benefits obtained from parallelization.

Implementation and data dependencies When the problem partitioning is done, all communication and synchronization points are determined and the high level parallel algorithm is designed, a programmer might start the actual implementation. Here a programmer will run into other types of problems. Consider two functionally equivalent code samples below.

```
for (int i=1; i<n; i++) {
  a[i]=a[i-1]+1;
}
```

Listing 2.1: Non-parallelizable loop with planted loop-carried data dependence.

```
for (int i=0; i<n; i++) {
  a[i]=a[0]+i;
}
```

Listing 2.2: Parallelizable loop free of any data dependencies.

The actual shape of the code can break its parallelization by introducing fake (not required by the algorithm) dependencies an average state-of-the-art compiler cannot tackle.

Performance analysis and tuning One needs to know where the program's hotspots are. Hotspots are the places where most of the real work is done. The majority of programs spend most of the CPU time in a few places. The task of a programmer is to find those places and concentrate all parallelization and optimization efforts there. Finding hotspots might be difficult before the programmer has the whole program implemented. Modern hardware architectures have a multi-level memory hierarchy, memory

data prefetchers, TLBs, out-of-order execution, etc. It might be surprising how the actual program execution performance differs from the one inferred from the algorithm. Profilers and other analysis tools can help here [94].

Finally, all the above challenges are interrelated and very often depend on each other. The parallel software development process can go iteratively with numerous dead ends and redesign efforts. With a long research history into the topic, all these problems are still actual to this day.

2.2.2 Limitations of automatic techniques

Given the difficulty of manual software parallelization, there have been numerous ongoing efforts into various automatic parallelization tools. The vast amount of legacy sequential software developed over the last few decades further exacerbates the need. Automatic parallelization refers to converting sequential code into multi-threaded and/or vectorized code in order to use multiple processors simultaneously in a shared-memory multiprocessor (SMP) machine.

There are various tools available to a programmer for automating the task of software parallelization. We present an overview of the field in the Section 3.1. Parallelizing compilers are the most widely used nowadays. Automatic parallelization tools can be classified into two types:

Fully Automatic The compiler analyzes the source code and identifies opportunities for parallelization. The analysis includes identifying inhibitors to parallelism and possibly a cost weighting on whether or not the parallelization would improve performance [27]. Loops are the most frequent target for automatic parallelization [12].

Programmer Directed Using compiler directives or possibly compiler flags, a programmer explicitly tells the compiler how to parallelize the code. These directives and flags may be also used in conjunction with some degree of automatic parallelization. The most common compiler-generated parallelization is done using on-node shared memory and threads (such as OpenMP [22]).

If one starts with an existing serial code and has the time or budget constraints, then automatic parallelization may be the answer. However, several important caveats apply to automatic parallelization.

Performance Performance may degrade.

Limitations Limited to a subset (mostly loops) of code.

Effectiveness May not parallelize the code, i.e. it may be ”over-conservative” or the code is too complex.

To clearly illustrate the problems an automatic software parallelization faces we conducted several experiments with the suite of NASA Parallel Benchmarks (NPB) [51]. These benchmarks target the performance evaluation of highly parallel supercomputers. Consequently, the suite has a great amount of inherent parallelism and is supposed to be easily parallelizable. NPB benchmarks do not provide the exact implementation, they rather specify what should be computed and how. We used Seoul National University's (SNU) implementation [57] of NPB benchmarks. SNU NPB implementation comes in two versions: sequential legacy C implementation and the one parallelized with OpenMP pragmas. The main data structure used everywhere in the suite is a simple flat array. Numerous loop nests operate over arrays and compute simple reductions.

For these experiments we used a desktop Ubuntu 18.04 machine with installed Intel C/C++ Compiler (ICC) 18.0 and measured the running time of benchmarks with the help of UNIX time() utility. To minimize the errors, we ran experiments several times and took the mean average. The machine has 4 Intel Core i5-6500 CPUs with 3.20 GHz frequency and vectorization support up to AVX2. The RAM is 16Gb.

The first experiment we conducted was aimed at assessing the effectiveness of the state-of-the-art automatic parallelization. The experiment is platform agnostic as long as the target supports vector instructions and parallel primitives and can be reproduced on any such platform. We took the Intel C/C++ Compiler (ICC), configured it for the most aggressive parallelization (*-par-threshold0*), i.e parallelize all parallelizable code independent of its potential cost weighted profitability. Also, we configured ICC to do all enabling loop transformations (*-O3* flag) before the actual parallelization (*-parallel* flag) and vectorization (*-vector* flag). In other words, the experiment measures the maximum parallelization coverage the best state-of-the-art compiler can achieve on embarrassingly parallel problems, which still represent the real-world code. Our results show a significant potential for improvement. Among all 1415 SNU NPB loops, 980 are truly parallelizable, but the ICC compiler manages to find only 812 parallelizable cases and misses 168 loops. Table 2.1 shows the classification we conducted by manually examining the ICC parallelization reports as well as looking at the source code of the benchmarks. The biggest problems are statically unknown pointers, which might potentially overlap at the running time, as well as other statically unresolvable dependencies. There are some unrecognized reductions as well as loops that can be parallelized with prior array privatization and function inlining. In 60 cases ICC could

reason	num	reason	num	reason	num
unrecognised reduction	18	**array privatization**	7	**AA conservativeness**	60
unknown iteration number	7	**static dependencies**	46	**too complex**	22
uninlined calls	4	**other**	4	**total**	168

Table 2.1: Classification of loops missed by Intel Compiler for various reasons.

not parallelize the loop due to pointers and alias analysis conservativeness. Static dependencies are largely indirect array references.

While parallelization coverage is important it is not the primary goal. A parallelized loop might not make a significant contribution to the total running time of the application. We should strive to parallelize those loops, which are on the critical paths and hot spots. And finally, only the running time is the ultimate parallelization performance measure. Given that, we conducted a further experiment. We used the same machine and compiled SNU NPB benchmarks with different sets of ICC automatic parallelization options. Figure 2.1 illustrates the running times of resulting codes. Bars marked as serial (s) show running times of original legacy C sequential versions. Bars marked as omp (o) show running times of an expertly manually parallelized versions. Other bars show running times of versions produced with various combinations of ICC compiler options (vectorize, parallelize or do both), as well as complete cases of parallel OpenMP versions, which have been additionally parallelized and vectorized by the ICC compiler. One can see that the best performance is still attributed to benchmark versions, which have been expertly parallelizad by their developers. Vectorization and parallelization help parallel versions a bit, but not that significantly and the profits can be neglected altogether. When we automatically vectorize serial versions we get a little improvement, but when we try to automatically parallelize them we get striking slowdowns on some benchmarks. Overall, automatic vectorization gives us a tiny 1.1x running time improvement in the geometric mean compared to 1.73x of manual parallelization. And the automatic parallelization results into 0.79x slowdown in the geometric mean across all the benchmarks.

Our machine learning based loop parallelization assistant [79] we propose in Chapter 4 extends parallelism recognition capabilities of the Intel C/C++ Compiler (ICC) by learning the loop parallelizability property and making predictions

regarding it with an acceptable false positives rate. These predictions cover most of the cases missed by ICC. Moreover, our assistant helps to reach the best possible manual expert performance faster by guiding the programmer towards to the most fruitful code segments to parallelize.

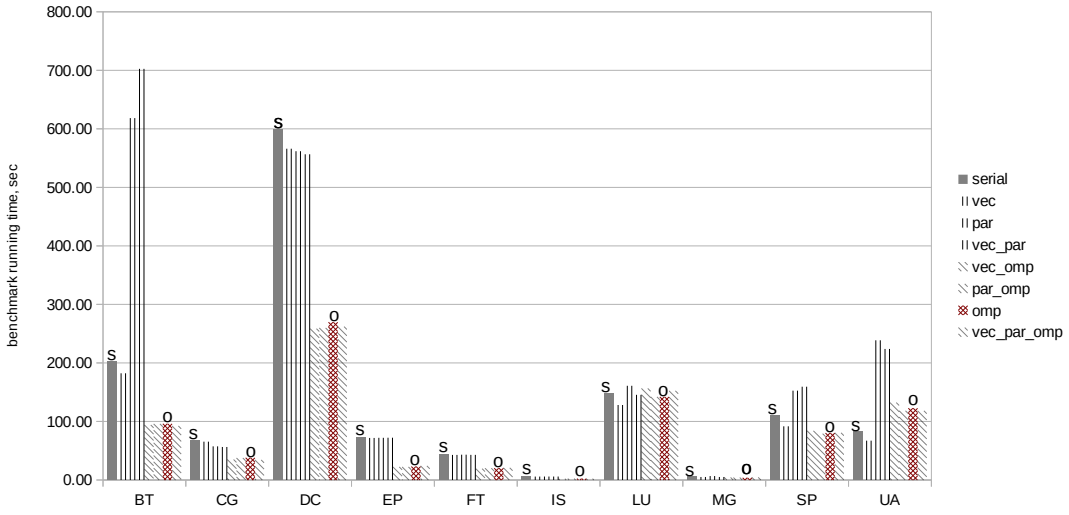

Figure 2.1: The running time of various NPB benchmarks versions.

2.2.3 Limits of machine learning based methods

Correctness is the most important property of the code. Although important, the running time or any other type of code performance characteristic is not always vitally critical. As all machine learning based techniques have always been characterized by their inherent and ineradicable errors [59], the field of compilers and especially the problem of automatic parallelization have never been the primary targets for these methods. Section 3.2 discusses the relevant work in more details.

The application of machine learning based methods to the problem of software parallelization has not yet found a widespread practical utility. Mispredictions regarding the code parallelizability can lead to a broken dependency property and thus incorrect program execution. Nonetheless, there have already been works on predicting loop parallelizability, like the approach of Fried *et al.* [58]. Fried *et al.* train a supervised learning algorithm on code hand-annotated with OpenMP parallelization directives to create a loop parallelizability predictor. These directives approximate the parallelization that might be produced by a human expert. Fried *et al.* focus on the comparative performance of different ML algorithms and studies the predictive performance that

can be achieved on the problem and does not produce any practical application.

In Chapter 4 we describe a practical ML-based loop parallelization assistant [79]. Contrary to Fried *et al.* [58] it uses static source code features. Moreover, we use a richer training set, which is not limited to expert OpenMP pragmas only, but additionally takes the information from the Intel C/C++ Compiler. While Fried *et al.* focus on the comparative performance of different ML algorithms and study the possibility to learn loop parallelizability property, we do the same, but additionally, we contribute a practical assistant capable of ranking loop candidates in their order of merit.

2.3 Data-Centric Parallelization (DCP) problem

The problem of data-centric parallelization (DCP) is the central motivation for the concept of computational frameworks we propose in Chapter 5. As it has already been stated the problem of software parallelization is multifaceted. There is a vast range of lower level technical issues, which can turn a perfectly parallelizable at a higher level computation into a non-parallelizable implementation. For example, in Section 2.2.2 we showed that the main reasons of Intel Compiler failures on SNU NPB benchmarks are alias analysis conservativeness, uninlined function calls and statically unresolvable dependencies. These reasons do not close the set of all possible parallel algorithm implementation failures. Listings 2.3 and 2.4 clearly illustrate a yet another potential implementation failure.

```
int a[1024];
for (int i=0; i<1024; i++) {
  a[i]=a[i]+1;
}
```

Listing 2.3: Parallelizable loop operating on what is clear to compiler a **linear array**.

```
struct Node* nptr;
for (p=nptr; p!=NULL; p=p->next) {
  p->value+=1;
}
```

Listing 2.4: Non-parallelizable loop operating on what programmer knows is a **linked-list**.

The above code snippets present two alternative implementations of the same simple and embarrassingly parallel computation. We increment all sequence elements by one. Listing 2.3 implements the sequence with a regular array linearly laid out in the memory. Listing 2.4 chooses a linked list as an implementing data structure. While in an array-based implementation the compiler knows all element addresses statically

and can generate parallel code in advance in the linked list based implementation element addresses can be resolved only dynamically, which leads to a source code non-parallelizability. Furthermore, compiler does not know what kind of a pointer-based data structure we iterate over.

The data-centric parallelization problem is how to automatically recognize what kind of a data structure is used in the code: is it a tree, a linked-list, a directed acyclic graph? The DCP problem is not solved yet. Automatic methods are limited in their recognition capabilities to relatively simple code bases such as libraries of well-known data structures. Automatic transformation is even harder. The most successful methods rely on the dynamic analysis of memory graphs. Static techniques such as shape analysis are undecidable and highly conservative and might not finish in a reasonable time for the real software projects. The Section 3.3 gives a comprehensive literature review on the topic.

2.4 Imperative and functional programming

Programming languages can be classified by different *programming paradigms* they support. Among the most general classifications are *imperative* and *declarative* programming languages.

Imperative programs are written in a form of instruction sequences, which read and write the *state* of a program. The concept of state is the main characteristic of the imperative programming paradigm. Instruction sequences can be structured in various ways. In *procedural programming* paradigm instructions are grouped inside procedures and functions. In *object-oriented programming (OOP)* paradigm instructions are grouped with the data they operate on inside objects of various types or classes. Programs are built either out of various procedures calling each other and exchanging the data or on the interaction of objects of various types. Imperative programs specify the exact sequence of steps to take in order to compute the final result.

Declarative programs do not specify the exact sequence of steps and state updates a program needs to do to get the desired result. Declarative programs declare the properties of the desired result. The properties can be specified as a set of constraints like in *constraint programming* or a set of linear inequalities like in *linear programming*. *Functional* programming is another subtype of declarative programming. In functional programming, the final result is specified as a sequence of stateless function evaluations, which form a tree of expressions. Among the most common constituents are

functions like *map*, *reduce*, *fold*, etc. Functions can be passed as arguments and returned from other functions ultimately composing bigger programs.

Functional programming is sometimes treated as synonymous with purely functional programming, a subset of functional programming that treats all functions as deterministic mathematical functions, or pure functions. When a pure function is called with some given arguments, it will always return the same result, and cannot be affected by any mutable state or other side effects. This is in contrast with impure procedures, common in imperative programming, which can have side effects (such as modifying the program's state or taking input from a user). Proponents of purely functional programming claim that by restricting side effects, programs can have fewer bugs, be easier to debug and test, and be more suited to formal verification.

There are no universally optimal programming paradigms and languages. Some languages are more convenient and suitable for one sort of problem, some languages are better at tackling other problems. For example, functional languages are more convenient in addressing certain domains such as R for statistics and financial analysis. Imperative languages are certainly better for simulations and other state-based scientific computations. For that reason, major languages are often multi-paradigm to cover a potentially larger set of problems. Largely imperative C++ language included support for functional programming with its newer standards starting from C++11.

2.5 OOP and software design patterns

2.5.1 Object-Oriented Programming (OOP)

Object-oriented programming (OOP) is arguably the most widely used programming paradigm nowadays. It is supported by almost all major programming languages. At the very essence, in OOP computer programs are designed by making them out of objects that interact with one another. Object interactions are very close to the human level of reasoning and logic and thus the paradigm fits quite naturally to a human developer.

Objects are instances of different types or classes in OOP terminology. Classes are object specifications. They specify the data objects contain (like an integer *age* field for an object of class *Person*) and the methods used to operate on the data. Classes define the public part of objects as well as their internal implementation. Object-oriented

(OO) languages provide a rich set of facilities and features to build programs.

Encapsulation is used for protection against object misuse and unintended outside interference: data and methods concerned with internal workings are declared *private*, while those designed to form an outward appearance are declared *public*. This facilitates code refactoring, for example allowing the author of the class to change how objects of that class represent their data internally without changing any external code. It also eases debugging by better localizing functionality and thus possible bugs.

Dynamic dispatch is the responsibility of the object, not any external code, to select the procedure to execute in response to a method call, typically by looking up the method at run time in a table associated with the object. This feature allows a programmer to write general code, which works with abstract interface methods and leaves the exact method resolution to be made during the running time of a program.

Dynamic dispatch is closely related to the technique of *inheritance*. Inheritance allows classes to be arranged in a hierarchy that represents ”is-a-type-of” relationships. Inheritance can be of two types: interface and implementation inheritance. The first one allows a parent class to require its descendants to stick to the same interface. A common interface allows the objects of different classes from the same hierarchy to be operated on by the same type agnostic code. The latter is called a *polymorphism*. The user code can be more concise and abstract. The call of the same method on the parent class or one of its descendants can result in a varying behaviour.

Features that are described above do not close the set of all available OOP techniques and mechanisms. These are just the main features we rely on in our project of computational frameworks and are the most important to know.

2.5.2 Software design patterns

The presence of all the above features makes OOP languages extremely rich with various facilities. That creates a vast design space for software architects and turns the OOP software design into an art. *Software design patterns* [17] are reusable solutions to common design problems in OOP. Design patterns have been well tested and are proven to be the most reliable and elegant solutions for the design problems they target.

Design patterns live at an intermediate level between the exact algorithm and a programming paradigm and are language agnostic. Design patterns specify and document solutions to common design problems. These solutions consist of a set of classes, ob-

jects, and the ways they interact with one another. It is agreed to classify patterns into 4 distinct categories: creational, structural, behavioral, and concurrency patterns. Here we are going to mention only those patterns, which are the most relevant to our project of computational frameworks.

The *command* pattern is a behavioral pattern, where a command object is used to encapsulate all the information needed to perform an action or trigger some event at a later time. The other participants are the receiver, invoker, and client. The central ideas of this design pattern closely mirror the semantics of first-class functions and higher-order functions in functional programming languages. Specifically, the invoker object is a higher-order function of which the command object is a first-class argument.

One can see the command pattern used in combination with the *chain of responsibility pattern*, which is a design pattern consisting of a source of command objects and a series of processing objects. Each processing object contains logic that defines the types of command objects that it can handle; the rest are passed to the next processing object in the chain. In a variation of the standard chain of responsibility model, some handlers may act as dispatchers, capable of sending commands out in a variety of directions, forming a *tree of responsibility*. The chain of responsibility pattern promotes the idea of *loose coupling*, where the system consists of multiple components, which know little or nothing about the definitions of other components.

Another very relevant pattern is a *template method* pattern. The template method is a method in a superclass, usually an abstract superclass, and defines the skeleton of an operation in terms of a number of high-level steps. These steps are themselves implemented by additional helper methods in the same class as the template method. The helper methods may be either abstract methods, for which case subclasses are required to provide concrete implementations, or hook methods, which have empty bodies in the superclass. Subclasses can (but are not required to) customize the operation by overriding the hook methods. The template method intends to define the overall structure of the operation while allowing subclasses to refine, or redefine certain steps.

The *visitor* pattern is a way of separating an algorithm from an object structure on which it operates. A practical result of this separation is the ability to add new operations to existing object structures without modifying the structures. In essence, the visitor allows adding new virtual functions to a family of classes, without modifying the classes. Instead, a visitor class is created that implements all of the appropriate specializations of the virtual function. The visitor takes the instance reference as input and implements the goal through double dispatch.

Software design patterns can be more down to the language like the *curiously recurring template pattern (CRTP)*, which is an idiom in C++ where class X derives from a class template instantiation using X itself as template argument.

Software design patterns specify and document the best practices to solve various commonly reoccurring problems. In the end, it is the experience, mastery, and ingenuity of a programmer which determine the final software design.

2.6 Algorithmic skeletons and Parallel design patterns

An overview paper [83] presents the concepts of *algorithmic skeletons* (or *parallelism patterns*) and *parallel design patterns* and tracks the way how these concepts have permeated the mainstream parallel programming community. We heavily relied on this work to prepare the material below.

In the last two decades, the general-purpose computing industry has made a massive shift towards parallelism. The number of various parallel hardware architectures available on the market has substantially increased. The change is not just quantitative: the number of parallel CPUs and CPU cores has increased, as well as the diversity and heterogeneity of parallel computing systems [56]. Modern systems feature various accelerators such as GPUs integrated with networked parallel CPU clusters into rather heterogeneous and complex computing environments.

The already difficult task of parallel programming (see Section 2.2.1) has become even more challenging, and the need for adequate programming models and frameworks to ease the job of parallel application programmers has become acute. Hardware architectures become pervasively parallel and grow in their complexity. There is an ongoing research effort that addresses the ease of programmability of the next-generation computing systems. Various structured parallel programming models have been introduced to tackle the problem.

Analogous to structured sequential programming, where *if-then-else*, *switch statement*, etc. have replaced unstructured *goto*-based programming style, parallel structured programming models ban certain programming practices. Concurrency can only be expressed and orchestrated through well-established parallel ”forms” and structured compositions of those. Structured parallel programming fosters not only more maintainable code but also makes it more parameterizable and predictable in terms of performance behavior.

Relatively low-level standards like POSIX threads [84] no longer meet all emerged

requirements to harness underlying hardware complexity and diversity and at the same time provide a convenient abstraction to application programmers. A number of further "de facto standards" like OpenMP (Open Multi-Processing) [87] for shared memory architectures, CUDA (Compute Unified Device Architecture) [93] and OpenCL (Open Computing Language) [89] for general-purpose GPUs, MPI (Message Passing Interface) [88] for distributed systems have been designed and introduced into the industry. Furthermore, the key industry players proposed other higher-level frameworks such as Intel TBB (Threading Building Blocks) [90] or Microsoft PPL (Parallel Patterns Library) [92]. Arguably, these models encompass the key results from academic research in the area of parallelism patterns.

Algorithmic skeletons or Parallelism Patterns The algorithmic skeleton (parallelism pattern) concept was introduced into academia by Murray Cole in his PhD thesis in the late '80s [7]. The algorithmic skeleton concept is intended to address the difficulty of programming parallel hardware architectures by separating two concerns: expressing application's parallelism through a composition of well-established and well-known primitives or algorithmic skeletons and designing and developing libraries of these primitives on a system programming side. Since the inception of the concept, a number of various algorithmic skeletons have been proposed. Among the most well-known patterns are *map*, *pipeline*, *reduce*, *scan*, *stencil*, *divide and conquer*, *task farm*, etc. More complex patterns can be composed of the basic ones. Algorithmic skeletons are higher than other parallel programming models and thus more human-friendly and portable. A more detailed description can be found in the book on structured parallel programming [54]. For example, a *pipeline* algorithmic skeleton can be defined like this:

Definition A *pipeline* is a chain of parallel processing entities arranged in a way so the output of each entity is the input to the next one. Suppose the pipeline consists of n stages described by functions $f_1, ..., f_n$. Then, for each data x in the input stream, the functions are applied consecutively to produce the final output $f_n(...(f_2(f_1(x))))$. When the stage f_j is done processing its input x_i, it can start to process the next input element x_{i+1} in the data stream. The next stage f_{j+1}, in turn, starts processing $f_j(x_i)$ data now in parallel. There are must be no race conditions between stages.

The above description presents a very general high-level specification that frames a parallel computation and leaves some space for creating various versions of the definition. An example of a more constraining definition can be found in the work [76] (see Section 3.4), which requires functions $f_1, ..., f_n$ to be pure, i.e have no side-effects.

Although, the violation of this requirement does not break pipeline parallelizability, as long as there are no race conditions between the stages.

Parallel design patterns The concept of software design patterns (see Section 2.5) has migrated to the world of parallel programming in '00s. A design pattern is a "recipe", that addresses some specific computational scenario. This recipe describes a problem together with the best-known solution. Parallel design patterns are described in the space of 4 dimensions: finding concurrency (i.e what parallelism can we exploit?), algorithm design (finding suitable algorithms), implementation structures, and execution mechanisms (multi-core CPUs, vector instructions, etc.). The table 2.2 below illustrates an example of *pipeline* parallel design pattern.

Name	Pipeline
Problem	Computation is organized in stages, over a large number of independent data sets
Examples	Assembly line, CPU instruction fetch/decode/execute cycle, video frame processing with multiple filters, etc.
Features	Parallel stage computation, input/output ordering, etc.
Implementation	Set up a chain of parallel activities, each one processing a stage, receiving input data from the previous stage, and delivering results to the next stage through proper single-producer single-consumer message queues.
Sample implementation	A sketch of MPI-based code computing a pipeline

Table 2.2: Sketch of sample pipeline parallel pattern specification (the whole pattern may easily take tens of pages to be properly described [54]).

Parallel design patterns specify computations on a very high level, leaving the whole software design, business logic implementation, and system-level parallelization to an application programmer. While, algorithmic skeletons usually come as libraries that implement low-level system side of parallelization behind a convenient API, alleviating the task for an application programmer.

Algorithmic skeletons and parallel design patterns themselves are largely well-established and are not at the cutting edge of the research effort. At the same time, the fifty years of parallel programming have generated a substantial amount of parallel legacy code using lower-level, more hardware-specific, and, hence, less portable, less maintainable

code. There is a range of ongoing research efforts to automatically identify various parallelism patterns in a legacy code and substitute them with their rejuvenated modern counterparts. Unfortunately, refactoring tools are still premature and are not fully adopted by development centers yet. Most of them are human-supervised, where a developer is responsible either for approving or doing the final manual transformation of the code. Although these tools relieve the burden of the source-to-source transformation, this process remains semi-automatic. The Section 3.4 provides a literature review on the subject.

2.7 Computational frameworks

The idea of computational frameworks we propose in this thesis (see Chapter 5) requires some clarification regarding its difference from algorithmic skeletons and its relationship with software design patterns and programming paradigms. The detailed discussion of computational frameworks is presented in Chapter 5. Here we introduce some clarifications to set a reader in advance.

2.7.1 A debate: computational frameworks vs. algorithmic skeletons

The difference between computational frameworks and algorithmic skeletons is very subtle. Let's start with considering *map* and *reduce* algorithmic skeletons. These skeletons specify a higher level computation to be applied to a set of elements. The elements can be arranged into an arbitrary graph, a sequence, a mesh, etc. These algorithmic skeletons are data structure agnostic as long as a certain computation can be applied to individual elements and produce a meaningful result. On the implementation side, data structures must provide proper iterators and conform to a required interface. A stencil algorithmic skeleton contains a notion of an underlying data structure. A stencil specifies how to combine adjacent elements of a regular data structure in higher-level parallel computation. A stencil could be computed on a 3D lattice, 2D mesh, or on a flat array. Although a stencil requires certain properties of an underlying data structure, it is not very specific.

The difference between well-established algorithmic skeletons and computational frameworks is best explained on a *fractal* (see Section 5.3.2), which is a form of re-

duction done over a tree data structure. The tree can be implemented as an array of pointers to leaves or a binary heap laid out on a flat array. Fractal is implementation agnostic in that respect, as it only requires the underlying data structure to be a tree and computation to follow a specified higher-level order. We cannot compute a fractal over a sequence or an arbitrary graph abstract data structure. Of course, a fractal can be implemented with a set of reductions, but it will no longer be a unified primitive.

While algorithmic skeletons are a higher-level concept that is either agnostic to an underlying data structure or implies some limitations while still allowing data structure variance within a certain space. The concept of computational frameworks is bound to a certain data structure over which it operates. In other words, it has both an algorithmic as well as data structure components.

In addition to fractal, the prototype library we implement to assess the concept also includes *map* and *reduce* algorithmic skeletons. This demonstrates the proximity of the two concepts and their compatibility. In our work, *fold* is also viewed as a computational framework and not an algorithmic skeleton, since it requires an underlying data structure to be a sequence. An *fold* algorithmic skeleton would only require of the underlying data structure to be recursive and, hence, is more higher-level, general and contains only an algorithmic component with minimal restrictions on the data structure. Both concepts are compatible.

2.7.2 Computational frameworks design

In Section 5.3.5 we describe the design of our prototype library [82]. The concept of computational frameworks and the prototype library we implement lie within the gap between functional and imperative programming paradigms: some problems contain computations, which are better expressed with standard functional concepts (like *maps*, *folds*, *reductions*, etc.), but at the same time require some state keeping. An example can be some scientific simulation. These problems lie at the boundary of functional and imperative programming. Our library provides a functional style interface and is developed in an object-oriented fashion.

The library implementing the concept of computational frameworks has been designed with some software design patterns in mind. While computational frameworks specify a higher-order algorithm, they leave a space for customization. The latter is implemented with the help of a template method design pattern, which also specifies the main computation and leaves some lower-level steps to be defined by a user. One can

see a similarity between the chain of responsibility pattern and the fold computational framework. The latter is also a visitor pattern in its way. While the backbone computation of our frameworks is implemented with a template method pattern, the command pattern is used to specify the user-defined custom part. Computational frameworks also promote the idea of loose coupling by keeping the computational procedure strictly decoupled from the side effects accumulation.

Computational frameworks relieve a programmer of some software engineering tasks (i.e software architecture design, writing lower-level system parallelization part) by providing off-the-shelf solutions for some specific scientific computation problems. Examples of these problems can be found in the Section 2.8.3.

2.8 Benchmark studies

In our projects we used three benchmark suites. We trained our machine learning (ML) based loop parallelization assistant [79] (see Chapter 4) with Seoul National University's implementation [57] of the NASA Parallel Benchmarks (NPB) [57]. The suite comes in two versions: sequential and parallelized with OpenMP. We used the latter to extract ML training labels. Moreover, the suite contains a load of various sorts of parallel loops, which makes it a good training set. For the data-centric parallelization (DCP) project we started with the SPEC CPU2006 benchmarks. The complexity level of that suite and the perspective we derived from the suite study led us to a simpler suite of Olden benchmarks. We used the latter for the project of computational frameworks (see Chapter 5). Below we provide descriptions of benchmarks, so a reader can develop a better feel for the problems we tackle and the code we work with.

2.8.1 NASA Parallel Benchmarks (NPB)

NAS Parallel Benchmarks (NPB) [51] target performance evaluation of highly parallel supercomputers. NPB are "paper-and-pencil" benchmarks, i.e they do not provide the exact implementation, but rather specify various computational problems at a higher level. There are various implementations. We used the one from Seoul National University (SNU) - SNU NPB [57].

There are 10 various benchmarks in the suite. Benchmarks perform various scientific computations. They solve various systems of linear equations, compute gradients, work with matrices (compute matrix transpose, inverse, etc.), solve differential equa-

tions, solve heat and diffusion equations on the mesh, compute 3d grids, etc. The main data structure used in all SNU NPB benchmarks is a flat multidimensional array. These arrays are processed in loops of various complexity levels including embarrassingly parallel loops like (*a[i] = b[i] + c[i]*), parallel copy and initialization loops (*a[i] = b[i]* and *c[i] = 0.0*), loops computing simple reductions, unrolled loops, multilevel nested loops and loops with uninlined function calls, as well as more complex loops with *if* and *switch* statements and loops with statically unknown iteration numbers and indirect array references (*a[i] = b[c[i]]*).

To the biggest part, these benchmarks are inherently parallel, but surprisingly their automatic parallelization with the best state-of-the-art Intel C/C++ Compiler (ICC) [3] results into a slowdown (see Section 2.2.2). Moreover ICC fails to recognize a lot of parallel loops. Our parallelization assisatant increases parallelization coverage and leads a programmer to achieve a manual expert level parallel benchmark performance faster.

2.8.2 SPEC CPU2006

The project of Data-Centric Parallelization (DCP) started with the feasibility studies of the SPEC CPU2006 benchmark suite. We studied the feasibility of the automatic data structure recognition techniques on these benchmarks. Although the benchmarks proved to be extremely complex for such techniques, these studies directed our further efforts and ultimately led to the concept of computational frameworks being an inseparable blend of data structures and algorithms. The key lessons we learnt from the SPEC CPU2006 benchmarks are the enormous complexity of the real legacy code and a very close relationship between data structures and algorithms. Let alone automatic techniques, it might take some weeks for an expert engineer to understand what a single benchmark is actually doing. Below we describe some of the benchmarks we looked at.

429.mcf The benchmark is derived from MCF, a program used for single-depot vehicle scheduling in public mass transportation. The benchmark operates with a complex network of nodes and arcs linearly allocated on the heap memory. Despite the simplicity of allocation, every node and arc has numerous pointers forming several object linking chains. Pointers are set in different places within the source code base (during allocation as well as during consecutive network structure updates), making the deduction of the actual data structure shape a task of grand complexity. The net-

work forms a spanning tree with several properties true of its nodes: every node has only one child pointer and if a node has several children, then the latter are connected through sibling pointers starting from the first child.

The tree data structure presents a high interest from the point of its recognition. But even a manual source code analysis and transformation requires a serious effort. Static automatic techniques seem infeasible, while dynamic ones seem to be a grand challenge.

456.hmmer Searches a DNA sequence database given a Profile Hidden Markov Model (HMM). The benchmark uses the Viterbi algorithm. The implementation works with four dynamic programming matrices allocated linearly as arrays. The algorithm walks either horizontally or diagonally along these matrices and computes reductions of maps. The computation is parallelizable and there has been successful work [45][44] doing it manually for specialized hardware.

Despite the complexity of its core function *P7Viterbi()*, the benchmark presents a very high interest for the application of our computational frameworks. Reductions of maps perfectly fit the purpose. We view it as future work.

400.perlbench This benchmark is a cut-down Perl interpreter, which implements the regular expression matching state machine. The benchmark processes the bitcode of a compiled regular expression instruction by instruction. Although instructions have the same size and are laid out linearly in memory, there might be branches and the whole processing happens in a linked-list offset-directed fashion. That requires sequential execution and is far beyond the capabilities of any existent techniques.

The benchmark is neither parallel nor a simple one. It makes no point to apply any recognition techniques here.

470.lbm The benchmark implements a Lattice Boltzmann Method (LBM) and is a relatively simple one (around 1400 LOC). The main underlying data structures the benchmark works with are the two 3D grids mapped onto a linear array space, which simulate incompressible fluids in 3D. The benchmark runs over arrays a specified number of time steps. Every array element represents a point from a 3D grid and consists of a number of velocity vector projections at this point. The values of these projections are being combined and mapped in a stencil fashion. The computation is highly parallel.

The benchmark operates with 3D grids laid out on regular arrays. The latter do not present a great deal of interest from the point of data structure recognition. Although, it would be interesting to try to recognize an algorithmic stencil.

2.8.3 Olden

Although the studies of SPEC CPU2006 benchmarks have proved their enormous complexity for the task of automatic data structure recognition, we acquired a good perspective and narrowed our research path to a much simpler suite of Olden benchmarks. Computations and algorithmic patterns present in Olden benchmarks have ultimately led us to the concept of *computational frameworks* (see Chapter 5).

Olden benchmark suite consists of 10 benchmarks. For our project we looked at 6 of those (*bisort*, *health*, *perimeter*, *treeadd*, *mst* and *tsp* benchmarks). The nature of different Olden benchmarks varies. Benchmarks health, treeadd, and perimeter perform essentially the same computational pattern, but for different problems. We call that pattern a fractal and it is basically a parallel tree growth and processing. Benchmarks tsp and mst solve 2 well-known graph problems namely travelling salesman problem (TSP) and minimum spanning tree (MST) construction. The other 4 benchmarks perform scientific numerical computations and are bigger, more complex, and less interesting from the perspective of data structure recognition.

bisort ***Definition*** *A sorted sequence is a monotonically non-decreasing (or non-increasing) sequence. A bitonic sequence is a sequence with* $x_0 \leq ... \leq x_k \geq x_{k+1} \geq ... \geq x_n - 1$ *for some k, or a circular shift of such a sequence.*

The sequence is implemented as a binary tree (recursive calls to the left and right subtrees). The algorithm is based on a sorting comparator network consisting of several layers. The network can be and is implemented in a divide and conquer way similar to that of a well-known merge sort. Sort() function is called on the left and right array halves recursively. The merging step of the merge sort algorithm is substituted with compare-and-swap step. The latter is possible due to input sequences required to be bitonic.

The benchmark is heavily based on pointers, tree swaps, and rotations. It presents an interest from the point of divide and conquer algorithm recognition. Static techniques are unlikely to handle the legacy source code of that complexity and style. Dynamic techniques might be able to see the binary tree.

health The health benchmark fits into the fractal computational framework the best and hence delivers the most promising performance results. Health benchmark does a simulation of the Columbian healthcare system and is based on a complete 4-ary tree of villages. One can view the tree as the hierarchical structure reflecting various levels of municipal divisions: the root is the capital, leaves represent villages

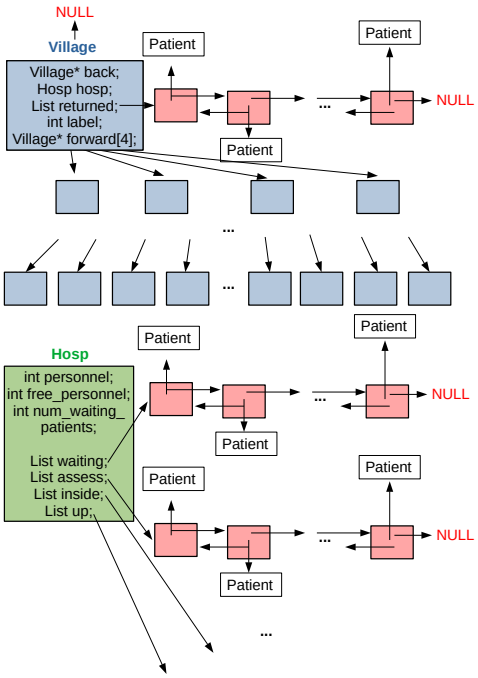

Figure 2.2: The *health* benchmark.

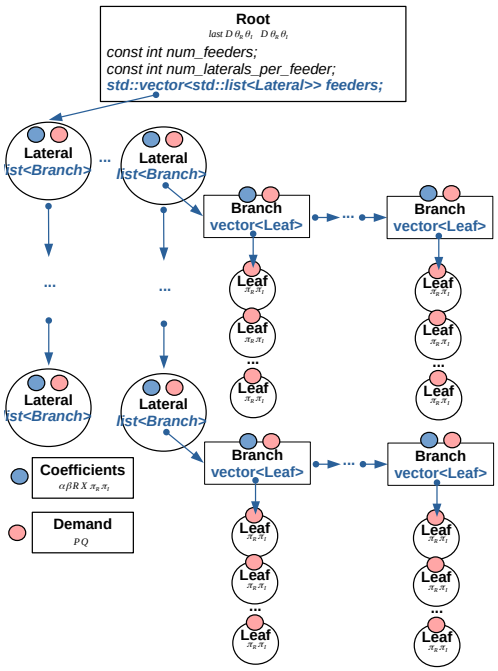

Figure 2.3: The *power* benchmark.

and the nodes on intermediate tree levels represent small towns and bigger cities. The closer to the root, the bigger the settlement and higher the level of hospital expertise. Every village has its own hospital. Figure 2.2 illustrates the tree.

Simulation starts at the tree root and goes down to the leaves. Some people fall ill in every village and when they do they go into the local hospital for an assessment and in case they are ill for treatment. If local staff cannot give an accurate diagnosis a patient goes up the tree to a bigger settlement with a higher staff expertise level. As the simulation goes the lists of patients waiting, under assessment, or inside the hospital for treatment grow and the benchmark state becomes bigger. So does the workload. The computation is parallel: all tree node child sub-trees can be simulated independently. Child nodes pass lists of patients to their parents and the latter take them to their local hospitals.

The benchmark operates with quad tree structures and does it in a highly parallel fashion. In our work, we call that pattern a fractal. The latter is a computational framework, i.e. the blend of an algorithm and a data structure. The whole structure can be aimed at for automatic recognition (not just a separate tree or an algorithmic skeleton, but both) even with static techniques.

perimeter The perimeter benchmark is another example of the fractal framework. The benchmark computes the perimeter of a ring (r=1024, R=2048). Figure 2.4 illus-

trates the process. The ring is placed onto a square, which is then being continuously and recursively divided into 4 equal sub-squares (southwest, northwest, southeast, and northeast). Only guided by the stop condition the process continues further. We divide the square further if it falls on the intersection with the ring boundary. If the square falls completely inside the ring or completely outside we stop the division. Division also stops, when the square area becomes less than a preset granularity (or equivalently the depth of the tree). The process can be represented as a growing unbalanced quad tree. Squares are the nodes of the tree. Squares inside the ring (all 4 square corners (x,y) are $r^2 < x*x+y*y < R^2$) are painted black, while those outside are painted white. In other words, the growth stops at black and white tree nodes and continues for those representing squares on the intersection with the ring. Finally, the resulting grid is being traversed to catch all flips of color. When the flip is detected we increment the perimeter counter by the square area adjacent to the flip boundary. The latter results into the final perimeter approximation.

The benchmark operates with quad tree structures and does it in a highly parallel fashion. This fits into the fractal pattern as well. The only difference from the health benchmark is the tree growth stop condition resulting into an unbalanced tree. Automatic fractal recognition seems harder, but still possible.

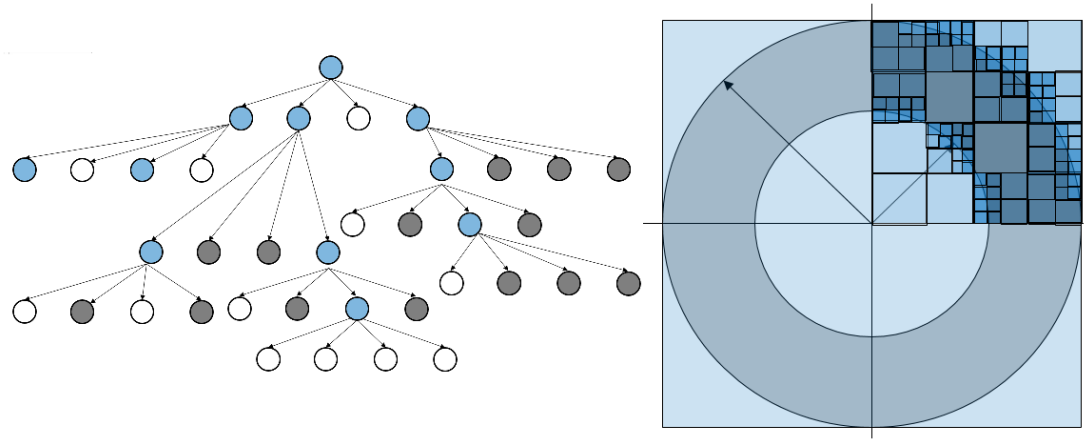

Figure 2.4: The **perimeter** benchmark. The computational pattern can be represented by an unbalanced quad tree.

power Another key benchmark for us is the power pricing computation benchmark. The benchmark is based on a composite hierarchical structure of C arrays and pointer-based linked lists of various objects: Root, Laterals, Branches, and Leaves. Figure 2.3 illustrates the data structure. The algorithm is basically composed of folds

and reductions. C arrays are reduced and linked lists are folded. The algorithm works recursively and starts with the Root::compute() call. The method accumulates the power demand Demand(P,Q) from all lists of Laterals and does a reduction to a final Demand(P,Q) value. Accumulation starts with the end of each lateral list. Each lateral accumulates its power demand from all its branches and the latter in turn accumulate their power demand from all their leaves. The leaves call optimize_node() method, which does a chunk of scientific computations (gradients, vectors, etc.), which compute P and Q. Finally, Root::compute() is called iteratively from a while loop of power_pricing_problem(). Iteration continues up until P and Q error becomes less than the required epsilon. Before every simulation step, the benchmark does an injection of coefficients into the structure. Every simulation step leaves computed side effects on the coefficients stored within various objects.

The power benchmark operates with sequences and linked lists. Sequences are reduced and linked lists are folded. These patterns correspond to and inspire our Fold and Reduce computational frameworks.

treeadd The treeadd benchmark is a very small and simple one. It does a recursive reduction on a pointer-linked binary tree. To create a workload the reduction is done repetitively inside the loop.

The treeadd benchmark is a straightforward example of the fractal computational framework. Although, the workload is too small for its effective application.

2.8.3.0.1 mst Minimum Spanning Tree (MST) benchmark does an MST weight computation of a complete graph. Computation is approximate and the algorithm looks like it might finish with the incorrect result. Nevertheless, the benchmark can be used as a computational workload. Graph is represented as a linked list of vertices. Every node in the list has a hash table of incident edges. Graph is complete: each vertex is connected to all other vertices in the graph (except itself). Algorithm repeatedly traverses the list of vertices and gradually accumulates the MST weight. On every traversal algorithm picks the node in the list to use as an input for the next traversal. In that sense, there is a cross iteration/traversal dependency. The code below summarises the benchmark.

```
vertex = list;
list = list->next;
while (num_vertices) {
    ret = traverse_vertex_list(vertex, list);
    mst_weight += ret.distance; // accumulate the final result
    vertex = ret.vertex; // next vertex to measure the distance
   against
    num_vertices--;
}
return mst_weight;
```

Listing 2.5: The main algorithm of mst benchmark.

2.8.3.0.2 tsp Travelling Salesman Problem (TSP). The benchmark generates a set of dots scattered on a 2D plane. Dots represent cities and are specified by their (x,y) coordinates. The TSP problem is to visit all the cities and return to the city of origin having passed the minimal distance. In other words, the algorithm returns a cycled sequence of cities, where the proximity of elements in the sequence in terms of order means their spatial proximity on the 2D plane. The algorithm's work resembles that of an insertion sort. The sequence is divided into 2 parts. Ordered part and unordered part. At first, the ordered part consists of just 1 element. On every iteration, the algorithm takes the next element out of the unordered part, finds it a pairing element inside the ordered part (with the minimal distance between them), and inserts the element into the ordered subsequence next to its pair. In the end, we get the sequence with the property that closest dots stand the closest in the sequence. The benchmark is based on a binary tree being transformed into a doubly-linked list. Every node of the tree represents a city located on a 2D plane with randomly generated (x,y) coordinates. The build_tree() method is written in a way to generate a uniform distribution of dots on the plane.

Chapter 3

Related Work

The topics of parallel computing, parallel software development, and software parallelization have a long and rich history. Historically parallel computers were programmed for doing various scientific simulations, particularly in the natural and engineering sciences. The limits of CPU frequency scaling and power consumption that came in 00' brought parallel computing to the area of desktop computers and general-purpose applications. Unfortunately, the legacy software did not adapt to these changes transparently and the field of software parallelization came to the spotlight.

Section 3.1 describes automatic parallelization. Automatically parallelizing compilers are largely limited to embarrassingly parallel scientific C and Fortran codes. To exploit the potential of upcoming multi-cores, and various heterogeneous systems it will be necessary to extend the scope of automatic parallelization to a broader class of programs containing complex pointer-based code and finding coarse-grain parallelism. Compilers cannot make such decisions in general as they cannot infer the higher-level semantics of the program. Sections 3.2, 3.3, 3.4 overview the research studying alternative parallelization strategies. Section 3.2 describes machine learning based methods. Due to the inherent statistical errors of these methods, the latter are relatively new to the area of software parallelization, where program correctness is paramount. The DCP problem has been introduced in Section 2.3. Section 3.3 provides a literature review on the subject. Lastly, Section 3.4 provides some references to works addressing the automatic recognition of parallel algorithmic skeletons and higher-level parallelizing program transformations. These problems are very challenging and not solved yet. Higher-level methods work in a manual or semi-automatic way requiring a programmer to approve or manually conduct the proposed transformation.

3.1 Automatic parallelization in compilers

There is a large body of work on automatic and semi-automatic parallelization of sequential legacy code [55], [80], [12], [27]. Years have been spent building various compiler infrastructures for research on parallelizing [27] and optimizing [21] compilers, e.g., SUIF (Stanford University Intermediate Format) compiler [19],[16],[97], Polaris [95], [13] and LLVM (Low-Level Virtual Machine) [1],[29]. The industry has its own well-known and well-established parallelizing compilers like Intel C/C++ Compiler (ICC) [3] or GNU Compiler Collection (GCC) project [96].

It is well understood how to parallelize scientific C and Fortran code with perfectly nested loops that operate over flat arrays [9], or even non-perfectly nested ones [20]. Furthermore, there are works on how to parallelize sequential loops across procedure boundaries [8],[10], or with some prior enabling transformations like array privatization [11]. These works, however, focus on loops over arrays. Some research efforts deal with larger code structures [30]. Decoupled software pipelining is a compilation technique to automatically recognize and extract thread-level parallelism from program loops by splitting the instructions of those loops into multiple smaller loops (pipeline stages) that execute in independent threads, and inserting dependence communication where necessary between these threads so that they remain synchronized [35], [40]. Software pipelining might require code duplication and eventually lead to code bloat that, if it is too large, can increase pressure on the cache memory and affect execution speed via a decrease in cache performance. Nevertheless, for loops with large trip counts on architectures with enough instruction level parallelism, the technique easily performs well enough to be worth any increase in code size.

Despite all the research efforts automatic parallelization remains largely limited to sequential scientific codes, where DOALL type loops operate over flat arrays with boundaries known at compile-time. There is a range of reasons that complicate the applicability of automatic parallelization techniques to wider areas. Dependence analysis is hard for code that uses indirect addressing, pointers, recursion, or indirect function calls because it is difficult to detect such dependencies at compile time. Loops often have a statically unknown number of iterations. Access to global resources might create race conditions and are difficult to coordinate in terms of memory allocation, I/O, and shared variables. Irregular algorithms that use input-dependent indirection interfere with compile-time analysis and optimization. All enumerated complications are very typical to the real-world code. As we demonstrate in Section 2.2.2 all above

problems can even materialize on a relatively simple and highly parallel suite of NASA Parallel Benchmarks (NPB) [51].

Speculative multithreading or thread-level speculation (TLS) techniques provide a workaround research direction to tackle the limitations of static program analysis. TLS techniques enable parallel execution of sequential applications on a multiprocessor by extracting speculative threads from serial code and submitting them for execution in parallel. At all times, there is at least one safe thread. While speculative threads venture into unsafe program sections, the safe thread executes code non-speculatively. Such techniques speculate the non-existence of dependencies [23] and speculative threads run under some assumptions on the values of input data. Should these assumptions prove wrong, then the state of speculative thread should be discarded. TLS is aware of the order in which such program sections would run in a single-threaded execution. Usually in TLS systems threads are assigned numbers, where the lowest one corresponds to the safe thread. As threads execute, the hardware checks for cross-thread dependence violations. For example, if a thread reads a variable and, later on, another thread with a lower number writes it, a true dependence has been violated. In this case, the offending reader thread is squashed and restarted on the fly. As speculative threads execute, they buffer all their memory state updates into some intermediate structures. If a thread is squashed, the buffer is flushed and all its memory state updates are discarded. If instead, all the thread's predecessors complete successfully, the thread becomes safe and it can commit its buffer into the main memory. If all assumptions prove to be correct the program can complete in a shorter time provided there were available hardware resources to schedule the threads efficiently.

TLS methods have their limitations. First, they require significant hardware support for handling thread contexts and detecting dependence violations in the memory subsystem [38],[24]. It is possible to implement these mechanisms in software, but then it might incur a significant amount of running time overhead, which would overwhelm any potential performance benefits from speculative-thread parallelism. Overall, speculative parallelization targets the inner program loops [39], [37], [38] and cannot handle coarse-grain parallelism, especially if the loops contain I/O or other system calls [47].

3.2 Machine learning in compilers

Correctness is the most important property of the code. Although important, the running time or any other type of code performance characteristic is not always vitally critical. As all machine learning based techniques have always been characterized by their inherent and ineradicable errors [59], the field of compilers has never been the primary target for these methods. Nonetheless, these techniques have found their application to some problems within the field.

ML in Compiler Optimization. Usually, machine learning based methods target problems, where a misprediction will only lead to a hampered performance and not a functional failure. For example, machine learning based methods have been used for finding the most optimal compiler optimization parameters like predicting the optimal loop unroll factor [42, 36] or determining whether or not a function should be inlined [31, 32]. These works are supervised classification problems and they target a fixed set of compiler options, by representing the optimization problem as a multi-class classification problem where each compiler option is a class. Recent works try to do scheduling and optimization of parallel programs for heterogeneous multi-cores. For example, Hayashi *et al.* [63] extracts various program features during compilation for use in a supervised learning prediction model aiming at the optimality of CPU vs. GPU selection. Evolutionary algorithms like generic search are often used to explore a large design space. Prior works [28, 33, 67] have used evolutionary algorithms to solve the phase ordering problem (i.e. in which order a set of compiler transformations should be applied).

Machine Learning and Parallelization. The application of machine learning based methods to the problem of software parallelization has not yet found a widespread practical utility. Mispredictions regarding the code parallelizability can lead to a broken dependency property and thus incorrect program execution. Nonetheless, there have already been works on predicting loop parallelizability, like the approach of Fried *et al.* [58]. Fried *et al.* train a supervised learning algorithm on code hand-annotated with OpenMP parallelization directives to create a loop parallelizability predictor. These directives approximate the parallelization that might be produced by a human expert. Fried *et al.* focus on the comparative performance of different ML algorithms and studies the predictive performance that can be achieved on the problem and does not produce any practical application.

3.3 Discovery: data structure recognition

The idea of automatic discovery of higher-level entities in programs is not a new one. This discovery problem is closely interlinked and entangled with alias analysis techniques [21] like points-to analysis [15]. The Points-to analysis is a variation of data flow analysis techniques. The final output is the sets of pairs of the form (p, x) (pointer variable p points to a stack-allocated variable x). These techniques are aimed at getting aliasing information regarding stack-allocated pointers.

The problem of understanding heap-directed pointers and heap-allocated linked data structures these pointers might point to is addressed with a family of static analysis techniques collectively known as shape analysis. Shape analysis techniques can be used to verify properties of dynamically allocated data structures in compile time. These are among the oldest and most well-known techniques. Three-valued logic [25][26] can be used as an example. The technique proposes the construction of a mathematical model consisting of logical predicate expressions. The latter correspond to certain pointer operating imperative language program statements. An abstract interpretation of these statements leads to the construction of sets of shape graphs at various program points. Shape graphs approximate the possible states of heap-allocated linked data structures and answer the questions such as node reachability, data structure disjointness, cyclicity, etc. The major limitation of these simplified mathematical models is the lack of precision high level of abstraction leads to. The problem of precise shape analysis is provably undecidable.

The work of [18] proposes a simplified and hence more practical implementation of shape analysis. Authors propose to use direction D and interference I matrices instead of complex mathematical models to derive shape information on heap-allocated data structures. The entry of direction matrix *D[p,q]* says if there exists a path from a node referred to by p to a node referred to by q. In other words, if we can enter a path within the data structure through p and exit through q. The entry of interference matrix *I[p,q]* says if the paths started from p and q are going to intersect at some point. Authors implement their technique withing McCAT compiler, which uses SIMPLE intermediate representation with a total of 8 statements (*malloc()*, pointer assignments p=q, structure updates p->*next*=q), which are capable of changing D and I matrices. Statements generate and kill entries in matrices. Moreover, they are capable of changing *Shape* attributes of pointers. The technique has been assessed on various benchmarks (bintree, xref, chomp, assembler, loader, sparse, etc.) from the era before the standard bench-

mark suites became available. The technique mostly reported shapes as *Trees* (be it a binary tree or a linked-list) or sometimes as *DAGs* or *Cycles* but with higher error rates in these last cases. The latter shows that the technique is imprecise and conservative.

One of the more recent techniques designed and developed by Ginsbach et al. is based on the pattern matching on LLVM IR level. The main idea is to specify computational idioms to be recognized in a domain-specific constraint-based programming language CAnDL [73]. Constraints are specified over LLVM IR entities such as instructions, basic blocks, functions, etc. The CAnDL language allows rapid prototyping of new compiler optimizations based on pattern recognition and its substitution with optimized versions of matched idioms. The language and its relatively fast backtracking constraint solver are capable of recognizing not only simple arithmetic idioms (thus performing different peephole optimizations), but more complex computations like general reductions and histograms [68], vector products in graphics shaders [74], sparse and dense linear algebra computations and stencils [74]. Having recognized these computational idioms the work [74] replaces them with a code for various heterogeneous APIs (MKL, libSPMV, Halide, clBLAS, CLBlast, Lift) and compares the resulting performance demonstrating an improvement over sequential versions and matching performance to hand-written parallel versions. The technique has been deployed on the sequential C versions of SNU NPB, the C versions of Parboil, and the OpenMP C/C++ versions of Rodinia demonstrating improved detection capabilities over the state-of-the-art techniques.

The other principally different technique has been recently proposed by Changhee Jung and Nathan Clark [41]. The authors developed a Data-structure Detection Tool (DDT) based on the LLVM framework. The tool instruments load, store, and call instructions within program binaries and gathers dynamic traces for sample inputs. The traces are used to recreate a memory allocation graph for program data structures. Call graphs are used to identify interface functions interacting with the built memory graph. DDT traces memory graph properties (number of nodes, edges, etc.) before and after interface function calls into another Daikon tool to compute dynamic invariants (the number of nodes in a memory graph decreases by 1 after every (delete()) interface method call, etc.). In the end, manually constructed decision tree is used to probabilistically match observed behavioral patterns against known data structure invariant properties. The technique has been deployed to recognize data structure implementations within standard libraries like STL, Apache (STDCXX), Borland (STLport), GLib, Trimaran achieving almost perfect recognition accuracy. Moreover, the technique has

been able to recognize linked lists in Em3d and Bh Olden benchmarks, along with red-black trees implementing vectors in the Xalancbmk benchmark.

There has recently been other published works on the application of dynamic techniques to the problem of dynamic data structure recognition [70][64]. The technique used in the DDT tool [41] makes an assumption, that all modifications and interactions with memory graphs representing data structures happen through a set of interface functions. That is not true when we deal with aggressively optimizing compilers, which may eliminate some code or inline some functions. The MemPick tool [64] searches data structures directly on a built dynamic memory graph by analyzing its shape. The graph is built with the help of the Intel Pin binary instrumentation tool during quiescent periods when pointer operations are absent. DSIbin tool [70] operates with the source code rather than program binaries. Instead of memory points-to graphs, it uses strands as primitives, which abstract such entities as singly-linked lists.

The work of Dekker [14] addresses the software design recovery problem in a completely different way. Contrary to the approaches described above, which operate on the IR and dynamic instruction stream levels, the work of Dekker operates at the level of an abstract syntax tree. Dekker's tool tries to compact the tree down to recognizable syntactic patterns by transforming it under a special grammar.

3.4 Discovery: algorithmic skeletons and parallelism patterns

Parallelizing sequential applications is a challenging and labor-intensive task, which requires a programmer to possess deep expertise in various fields. Parallelization can be done in numerous ways, due to many different ways in which an application can be parallelized and a large number of various programming models that can be used. The concept of algorithmic skeletons (parallelism patterns) (see Section 2.6) has been proposed to alleviate the challenging task and provide a programmer with a portable human-friendly model of parallel computations. There is a vast array of various frameworks and libraries that implement the concept. They vary by the programming language they target, distribution library used to implement parallel/distributed computations internally (like MPI [88], OpenMP [87], etc.), the sets of supported skeletons, and the allowed level and way of skeleton nesting, i.e composing more complex patterns of the basic ones; these include: RPL [66]; FastFlow [62]; Microsoft's Pattern

Parallel Library [92]; and Intel's Threading Building Blocks (TBB) library [90].

Although, the libraries of parallelism patterns alleviate the process of parallel software development, their use still requires a manual effort. Before using a particular library, a programmer has to invest time and effort into learning its target use and limitations. Furthermore, discovering places in sequential code where parallel patterns might be introduced is still highly non-trivial, often requiring expert manual analysis and profiling. There is a number of works addressing the task of automatic discovery of parallel patterns.

The work [76] proposes a static approach and develops Parallel Pattern Analyzer Tool (PPAT) capable of analyzing sequential C++ code statically to detect parallel patterns. The tool takes advantage of Clang library to generate an Abstract Syntax Tree (AST). It walks through the AST finding loop candidates to be analyzed for possible parallel patterns. For every loop, the tool checks a set of constraints specific to every target pattern. For example, for a loop to be a parallel pipeline, it must not write any global variables, pass no feedback between separate loop body stages, and there must be at least two stages to split the loop into. For farm pattern, the tool checks that the loop body has no RAW dependencies, no break statements, and writes no global variables. Authors evaluate the effectiveness and correctness of their approach using NAS [51] and Rodinia benchmark suites by comparing automatically detected patterns against the manual analysis. The authors observe that the pattern detection quality of PPAT is close to that performed by a human expert. Authors parallelize detected patterns manually demonstrating final performance comparable to expertly parallelized benchmark versions. Therefore, reducing the human effort in transforming sequential codes into parallel.

Some works take advantage of functional languages. For instance, István Bozó et al. [61] develop a tool that detects parallel patterns in applications written in Erlang. Compared to other languages, Erlang features make the detection process much simpler. Nonetheless, the tool requires profiling techniques to decide which pattern suits best a concrete problem.

The work [85] aims to highlight to a programmer code fragments, which could be replaced by calls to known parallel pattern library abstractions of map, reduce and their compositions in a legacy Pthreaded C/C++ code. The underlying technique is based on the analysis of dynamic data flow graphs (DDFGs) obtained during the execution of programs and thus is language-agnostic. The analysis uses constraint-based pattern matching to identify constrained DDFG subgraphs characteristic to the well-

established parallel patterns while employing heuristics that trade analysis time against completeness making the approach scalable. The tool and methodology demonstrate excellent effectiveness and accuracy on Starbench benchmarks by finding 36 out of the expected 42 instances of parallel patterns with a high accuracy (reporting actual patterns in 98% of the cases). Authors re-express the found patterns via a parallel pattern library, making code freely portable across CPU/GPU systems and performing competitively with hand-tuned implementations at zero additional effort.

There are also hybrid methods of parallel pattern discovery. The work [81] employs both static and dynamic trace-based analysis, together with hotspot detection. First, using running time profiling the methodology obtains a list of hotspot loop candidates, which pass through static analysis, leveraging the existing Pattern Analyzer Tool PPAT [76]. Independently from the static analysis, candidate loops are also passed through a new dynamic trace-based pattern detection mechanism. Finally, the user checks manually that the candidate detected patterns (from either analysis) are indeed applicable. The mechanism is evaluated on a number of representative benchmarks, demonstrating good accuracy, precision, and recall scores for map and reduce algorithmic skeletons. The scores are compared against the manual analysis.

As the sequential code gives the cleanest starting point for the introduction of parallel patterns, the work [86] studies how parallel legacy C/C++ pthreaded codes could be converted back to sequential version and ultimately restored to a modern parallel pattern based equivalent form. This work studies the specifics of parallel legacy pthreaded codes and proposes a novel methodology to accomplish the task. The restoration is conducted through a systematic application of a number of identified program transformations. Authors design and define a set of restorative transformations common to many legacy pthreaded codes. These transformations aim to (i) eliminate Pthread operations from legacy C/C++ programs; (ii) perform code repair, fixing any bugs introduced in (i); and, (iii) reshape code in preparation for parallel pattern introduction. The work targets only farm and pipeline patterns. The transformations presented in the work are intended as manual transformations. The implementation of these refactorings into a semi-automatic tool is envisaged as future work. Authors use the Intel TBB library to evaluate these transformations on a set of benchmarks and demonstrate that the removal of parallelism allows to manually derive structured parallel code that is comparable to the original legacy-parallel version in terms of performance while being more portable, adaptive, and maintainable. Additionally, authors record improvements in terms of cyclomatic complexity [5] and Lines Of Code (LOC) metric.

Chapter 4

Software Parallelization Assistant

4.1 Introduction

Parallel hardware is ubiquitous through the entire spectrum of computing systems, from low-end embedded devices to high-end supercomputers. Yet, most of the existing software is sequential. Despite decades of intensive research in automatic software parallelization [55], fully exploiting the potential of modern parallel hardware still requires a significant manual effort. Given the difficulty of the obstacles faced by automatic parallelization today, we do not expect that programmers will be liberated from performing manual parallelization in the near future [53].

This chapter introduces a novel parallelization assistant that aids a programmer in the process of parallelizing a program in the frequent case where automatic approaches fail. The assistant reduces the manual effort in this process by presenting a programmer with a ranking of program loops that are most likely to 1) require little or no effort for successful parallelization and 2) improve the program's performance when parallelized. Thus, it improves over the traditional, profile-guided process by also taking into account the *probability* of potential parallelization for each of the profiled loops.

At the core of our parallelization assistant resides a novel machine-learning (ML) model of loop parallelizability. Loops are compelling candidates for parallelization, as they are naturally decomposable and tend to capture most of the execution time in a program. Furthermore, focusing on loops allows the model to leverage a large number of specific analyses available in modern compilers, such as generalized iterator recognition [75] and loop dependence analysis [69]. The model encodes the results of these analyses together with basic properties of the loops as machine learning *features*.

The loop parallelizability model is trained, validated, and tested on 1415 loops from

the SNU NAS Parallel Benchmarks (SNU NPB) [50]. The loops are labelled using a combination of expert OpenMP [22] annotations and optimization reports from Intel Compiler (ICC), a production-quality parallelizing compiler. The model is evaluated on multiple machine learning algorithms, including tree-based methods, support vector machines, and neural networks. The evaluation shows that – despite the limited size of the data set – using support vector machines allows the model to achieve a prediction accuracy higher than 90%. The model improves over the ICC Compiler across the sequential C version of the SNU NPB suite by detecting 13% more parallel loops. Albeit this improvement comes at the cost of introducing *false positives*, where non-parallelizable loops are misclassified as parallelizable. However, the false positive rate in our evaluation is as low as 6.5%. We feel this is acceptable, as our parallelization assistant does not automatically restructure code, but leaves the parallelization decision in the hands of the programmers.

The parallelization assistant combines inference on the parallelizability model with traditional profiling to rank higher those loops with a high probability of being parallelizable and impacting the program performance. An evaluation on eight programs from the SNU NPB suite shows that the program performance tends to improve faster as loops are parallelized in the ranking order suggested by our parallelization assistant compared to a traditional order based on profiling only. On average, following the order suggested by the assistant reduces by approximately 20% the number of lines of code a programmer has to examine manually to parallelize SNU NPB to its expert-level speedup. Given the high level of effort involved in manual analysis, such a reduction can translate into substantial development cost savings.

4.1.1 Motivating example

Consider the sequential C implementation of the *Conjugate Gradient (CG)* benchmark from the SNU NPB suite. Table 4.1 shows the top three CG loops as ranked by the Intel Profiler (based on their execution time) and by our parallelization assistant (additionally taking into account their parallelizability). The ranked loops compose the same loop nest with `cg.c:296` being the outermost loop and the loop `cg.c:458` being the innermost. The profiler ranks their execution time in the nesting order, while our assistant highlights the same loop nest, but, crucially, the loops are **ranked in a different order** with the innermost `cg.c:458` coming first. Following the profiler ranking (second column in Table 4.1), a parallelization expert would concentrate on analyzing

Ranking	Profiler	Assistant	
	loop	loop	parallelizability
1	cg.c:296	**cg.c:458**	85%
2	cg.c:445	cg.c:296	29%
3	**cg.c:458**	cg.c:445	8%

Table 4.1: Comparison of the profiler and assistant rankings for the CG benchmark loops (limited to the top three loops).

the longest running outermost loop cg.c:296 first. Analyzing this loop turns out to be costly (it consists of 100+ lines of code) and unfruitful (it is not parallelizable due to inter-iteration dependencies and side effects, see Listing 4.1). This analysis would be followed by an equally unfruitful analysis of cg.c:445 being the middle loop in the nest.

In contrast, our parallelization assistant (two last columns in Table 4.1) ranks the innermost loop **cg.c:458** before its two enclosing loops cg.c:296 and cg.c:445, as it finds that the former has a significantly higher probability of being parallelizable (see last column in Table 4.1). Following the assistant's ranking, a parallelization expert would thus concentrate on analyzing the loop **cg.c:458** first. Analyzing this loop is inexpensive (it consists of only six lines of code, see Listing 4.2) and fruitful: its parallelization speeds up CG by a factor of 2.8, which is 70% of the speedup obtained by parallelizing the entire benchmark. Hence, following the ranking proposed by our assistant, a parallelization expert can achieve most of the available speedup in CG in a fraction of the time required by a traditional profile-guided parallelization process.

It would also be interesting to give a theoretical assessment of the possible CG parallel speedups with Amdahl's law [4]. The benchmark CG contains a number of parallelizable loops and some serial parts. The Intel Profiler reports that the main loop nest takes up 93% of CG execution time and contains a number of parallelizable loops inside, but the longest-running **cg.c:458** takes up roughly 90% of the total CG running time. It would be precise enough to assume, that 95% of CG running time is spent executing parallelizable sections of the code. With 90% of execution happening within **cg.c:458**. The Amdahl's law derivation would yield the following estimate $s = \frac{20}{1+1/m+18/n}$, where m and n factors characterize the speedup of the corresponding parallel parts. On the 4-core machine we used for the experiment, the factors in the formula would be around 4. So, the theoretical parallel speedups would be 3.5x for

```
for (it = 1; it <= NITER; it++) {
  ...
  if (timeron) timer_start(T_conj_grad);
  conj_grad(colidx,rowstr,x,z,a,p,q,r,&rnorm);
  if (timeron) timer_stop(T_conj_grad);
  ...
  printf("    %5d       %20.14E%20.13f\n", it, rnorm, zeta);
  ...
}
```

Listing 4.1: `cg.c:296`. The longest running loop in CG. The loop cannot be parallelized due to inter-iteration dependences and side effects caused by system calls.

```
// lastrow - firstrow + 1 = NA, NA is the workload parameter (S
   =1400,M=7000,L=140000,...)
#pragma omp parallel for default(shared) private(j,k,suml)
for (j = 0; j < lastrow - firstrow + 1; j++) {
  suml = 0.0;
  for (k = rowstr[j]; k < rowstr[j+1]; k++) {
    suml = suml + a[k]*p[colidx[k]];
  }
  q[j] = suml;
}
```

Listing 4.2: **`cg.c:458`**. Longest running loop in CG among those *that can be parallelized*. Parallelization pragmas are also added to demonstrate its parallelization.

the whole CG benchmark and 3x for the innermost loop only. The theoretical speedup corresponds to what we observe experimentally (3.25x and 2.8x).

4.1.2 Contributions of our Loop Parallelization Assistant

In summary, this chapter makes the following contributions:

- ▷ We introduce a machine learning model, which can be used to predict the probability with which sequential C loops can be parallelized (Sections 4.2 and 4.3);
- ▷ we integrate profiling of execution time with our novel ML model into a parallelization assistant, which guides the user through a ranked list of loops for parallelization (Section 4.4); and

▷ we demonstrate that our tool and methodology increase programmer productivity by identifying parallel loop candidates better than existing state-of-the-art approaches (Section 4.5).

4.2 Predicting parallel loops

We approach the prediction of parallel loops as a *supervised probabilistic machine learning classification problem*. Based on sequential reference applications and their manually parallelized counterparts, as well as Intel C/C++ Compiler's (ICC) parallelization reports, we create a data set of parallelizable and non-parallelizable loops. We extract loop features and use the data set to train a machine learning model, which links feature vectors describing the loops with their observed parallelizability. We then use the trained model as a probabilistic predictor: for each new loop, we determine its feature vector and then predict the probability of the loop being parallelizable [34]. For naturally probabilistic models like trees, we directly use the computed classification probabilities (fraction of parallelizable training samples in the leaf node). For the support vector machines classifier, we use Platt scaling to derive the probabilities.

In this section, we introduce the parallelizability model, whereas Section 4.3 presents a standard ML performance assessment including accuracy, precision, and recall scores. Descriptions and definitions of the machine learning techniques we use can be found in [59]. We used the *scikit-learn* library [49] for all ML related tasks.

4.2.1 Loop analysis & Feature engineering

For the purpose of machine learning, program loops are represented by numerical *feature vectors*. We derive these features using standard compiler analyses operating on the Program Dependence Graph (PDG) [6] of a loop. The PDG is a representation that captures both data and control information and is constructed using dependence analysis [27]. Figure 4.1 shows an example of a PDG for a simple loop, where SCC stands for *Strongly Connected Component*. We construct the PDG on a program's LLVM IR using standard dependence analysis [27]. In addition, we use *generalized loop iterator recognition* [75] to separate *loop iterators* from *loop payloads*. This enables us to define and extract features relating to each of those loop components. In total, we extract a set of 74 static loop features which are based on structural properties of the PDG and the types of instructions constituting it. Table 4.2 summarizes these

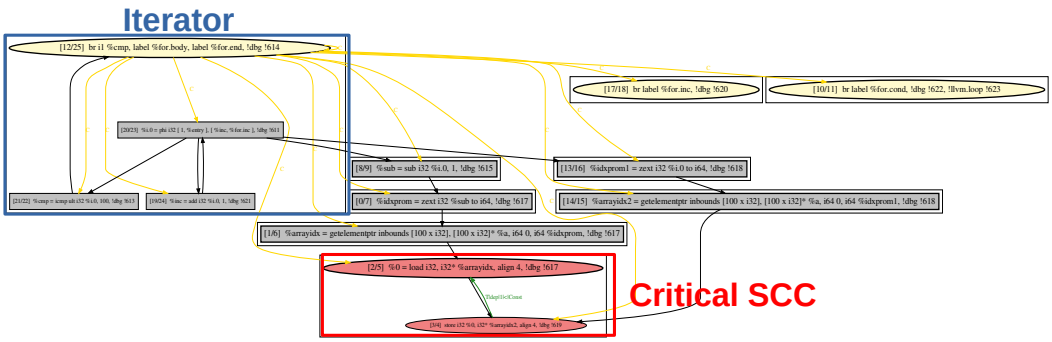

Figure 4.1: Example of PDG of a simple loop with a cross-iteration dependency.

features.

Many features have simple and intuitive motivations behind them. Loop proportion features are backed up by the fact that larger loops tend to be harder to parallelize. Complex iterators include non-trivial cross-iteration transitions (e.g. linked-list update), unknown iteration numbers, etc. Payload SCCs introduce cross-iteration dependencies. Cohesion features characterize how tightly components of loops are coupled together in terms of the number of edges between them. Loop dependence features count the number of edges in different loop parts as well as their types. Loop instruction features characterize the loop's instruction mix, assigning more importance to memory reads/writes, calls and branches. Non-inlined function calls usually prevent loop parallelization. Intensive memory work (memory read/write fraction features) complicates parallelization as well.

4.2.2 Feature extraction

To extract all devised loop features from SNU NPB benchmarks and get the train/test data sets we developed a tool based on the LLVM compiler infrastructure [1][29]. The tool is a set of LLVM function passes working on the SSA-based LLVM IR and can be found on GitHub [78]. The tool works by building data, memory, and control dependence graphs (DDG, MDG, CDG) and combining them into the final program dependence graph (PDG) [6] for all loops found in program functions. Once all graphs are built we run the search of SCCs [75] on them and recognise loop iterators. The final step is to traverse all these graphs computing devised metrics (ML features) and dump all that information into the file to be later fed into scikit-learn based ML scripts.

Feature groups	Features	Description
loop proportion	absolute size	number of LLVM IR instructions
	payload fraction	payload instructions / total loop instructions
	proper SCCs number	number of payload SCCs with more than 1 instruction
loop dependencies	number of PDG edges for different dependence classes: read/write order (*true*, *anti*, *output*), dependency type (*register*, *memory*, *control*), other (*cross-iteration*, etc.)	
loop cohesion	iterator/payload	$\frac{\text{edges between iterator/payload}}{\text{total loop edges}}$
	critical/regular payload	$\frac{\text{edges between critical/regular payload}}{\text{total loop payload edges}}$
loop instruction nature	numbers and fractions of different parallelization critical instructions (memory loads and stores, branches, calls, etc.)	

Table 4.2: Static features used for the characterization of loops.

4.2.3 Feature selection

The feature engineering task produced a quantitative description of program loops being characterised by feature vectors of length 74. To avoid overfitting, we discard irrelevant or redundant features using a pipeline of automatic feature selection methods from the scikit-learn library. First, we eliminate features with a low variance score, then we fit a decision tree-based model and select features with importance scores above a given threshold. After that, we repeatedly run *Recursive Feature Elimination, Cross-Validated (RFECV)* to improve accuracy, precision, and recall scores. This yields the final set of features. Table 4.3 presents the 10 highest-ranked features in the set. SNU NPB benchmarks contain a lot of uninlined function calls and, unsurprisingly, the amount of call instructions in the payload of a loop ranks the highest. Despite the absence of straightforward intuition behind cohesion metrics, they tend to correlate with loop parallelizability well. Loops heavy on memory writes also significantly affect the parallelizability property.

4.2.4 Model & Hyper-parameter selection

We evaluate several machine learning classification algorithms in our parallelization assistant, including tree-based methods like decision trees (DT), random forests (RFC),

Feature	Importance
payload call fraction	23.5
iter/payload non-cf cohesion	18.5
payload mem write fraction	6.1
loop absolute size	5.7
critical payload pointer access count	5.3
payload memory dependence count	4.0
critical payload non-cf cohesion	2.9
payload pointer access fraction	2.7
critical payload total cohesion	2.6

Table 4.3: Relative importance of static loop features, ranked by fitting a tree-based ML model.

and boosted decision trees (AdaBoost); support vector machine classifiers (SVC) and multi-layer perceptron neural networks (MLP). Section 4.3.1 shows that these models perform similarly with SVC and MLP performing slightly better. For each ML model, we use exhaustive hyper-parameter grid search and pick the grid node with the best cross-validation score on the validation set. The details of all ML pipeline stages are available in our repository [78].

4.2.5 Training data & ML model training

In order to train and test our ML model in a supervised way, first we need to provide it with the ”right answers” regarding loop parallelizability. In other words, we need to prepare a labelled training data set. For training our ML model we use a total of 1415 loops from the SNU NPB benchmark suite. Out of those loops, 210 have been annotated by (external) human experts with parallel OpenMP *pragmas*. Like [58] we use these annotations as labelled data to indicate parallelizable loops. However, the data is not complete. Human programmers strive to capture only coarse-grain parallelism and do not annotate every parallelizable loop. Hence, we augment the training data with the help of the ICC Compiler, which finds additional parallelizable loops. We combine the results into our final training set comprising a total 1415 loops, of which 995 are labelled as parallelizable. Then we use K-fold and Leave-One-Out Cross-Validation (LOOCV) methodologies to train and test our ML models.

To extract loop parallelizability labels from the Intel compiler’s optimization reports

we developed a parser [2], but the task has presented us with a number of technical challenges. Before ICC can actually parallelize or vectorize a loop, it applies a number of enabling loop transformations such as loop interchange, distribution, tiling, etc. These transformations help ICC to eventually uncover more parallelism. The detailed description of all these transformations can be found in [12]. Applied to a loop nest, these optimizations might significantly restructure and distribute the parts of a loop across the whole ICC optimization report. Moreover, ICC might parallelize only certain parts of transformed loop. In the end, we considered a loop to be parallelizable by the ICC compiler if the latter hasn't found any dependencies and either vectorized or parallelized it. In the case of distributed loops, all parts must be parallelizable for an original loop to be considered as such. For a final correctness we conducted a manual verification on top of automatically extracted results.

Table 4.4 presents a parsing report, which summarises the number of times ICC applied a certain optimization. The major cells are *parallel* and *icc*, which report the total number of truly parallelizable loops and the number of loops parallelized by ICC. As can be seen, ICC compiler does not exploit all the parallelism available in SNU NPB benchmarks. Section 2.2.2 presented a study of the reasons why ICC fails to parallelize certain loops. To obtain the data we used ICC 18.0. As we are interested in obtaining the most complete set of parallelization labels, and not in the optimal program running time, we instructed ICC to do the most aggressive parallelization, i.e. parallelize a loop, whenever it can be parallelized, ignoring ICC cost model and independent of the expected profitability of loop parallelization (*-parallel -vector -par-threshold0* options). To make ICC uncover all possible parallelism we ran it with a wide set of enabling loop transformations (*-O3* option) applied prior to the parallelization. All the options had a matching hardware support: we installed ICC on the Ubuntu 18.04 machine with 4 Intel Core i5-6500 CPUs having a vectorization support all up to AVX2.

4.3 ML predictive performance

In our work we employ 2 cross-validation (CV) techniques. We evaluate the overall predictive performance our trained ML model is capable of achieving on SNU NPB benchmarks using K-fold CV. To deploy our assistant against single benchmarks of the suite and assess its effectiveness (Section 4.5) we have to use a modified Leave-One-Out CV.

Labels		Intel Compiler (ICC)			
		Optimization		Parallelization	
loop	ranking	loop	ranking	parallel	
total loops	**1415**	distrs	34	parallel	653
parallel	**995**	fusions	214	vector	737
icc	**812**	collapses	58	parallel deps	535
openmp	**210**	tilings	27	vector deps	266

Table 4.4: Report on loop classification labels derived from expertly added OpenMP annotations of SNU NPB benchmarks and ICC optimization reports. Out of all 995 parallelizable loops the ICC discovered and parallelized only 812.

ML model	accuracy	recall	precision
constant	70.32	100	70.32
uniform	46.27	41.50	69.79
SVC	90.04	95.24	91.06
AdaBoost	86.96	92.92	89.06
DT	84.36	89.57	87.90
RFC	86.65	93.22	88.47
MLP	89.40	93.77	91.39

Table 4.5: Average predictive performance of different ML models measured with K-fold CV on the whole set of 1415 SNU NPB loops.

4.3.1 Overall model performance

Table 4.5 shows the overall predictive performance of different ML models measured with K-fold CV on the whole SNU NPB data set. Training and testing have been done for different values of K (5, 10, 15, 20, 25, 30) and the accuracy remains stable across the entire range. The same is true of recall and precision scores. We used the baselines (constant ”parallelizable” prediction and uniform) available in scikit-learn to compare our models against. The SVC model has the highest average accuracy and successfully manages to recall 95.24% of all parallel loops. The ICC Compiler succeeds in parallelizing 812 out of 995 parallelizable loops available in SNU NPB. Thus, on average SVC extends the ICC Compiler's parallelization capabilities to 948 loops. Figure 4.2 shows that out of the 10% of mispredictions that SVC makes, 65% are false positives. Hence, the average unsafe error rate is 6.5%. In this project, though we

devise a scheme, that protects us and makes these errors not critical.

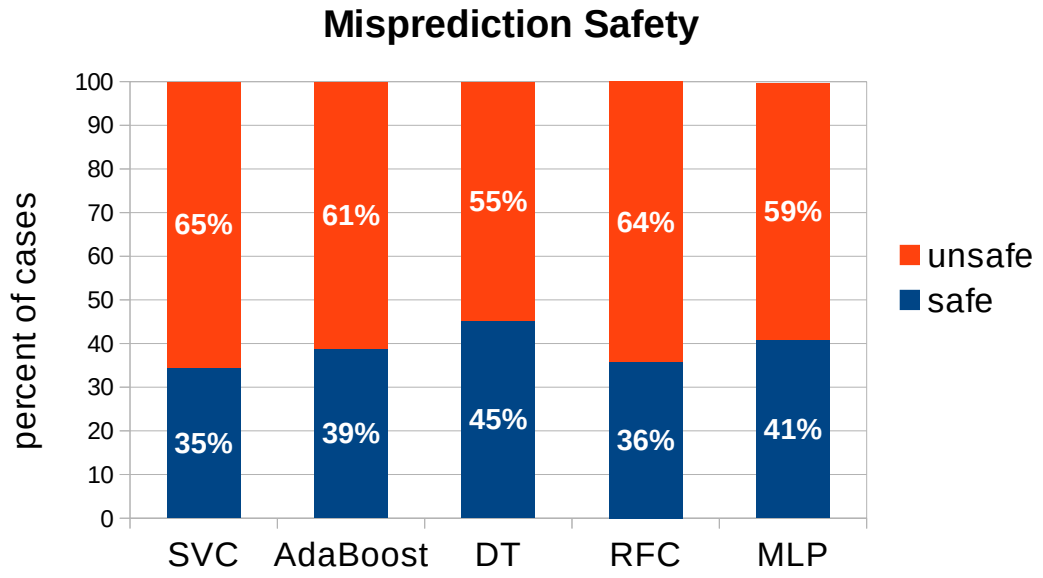

Figure 4.2: Breakdown of misclassification errors.

4.3.2 Model performance within assistant

Our proposed assistant (Section 4.4) is trained and tested using LOOCV rather than K-fold CV. Instead of treating the entire set of loops from all SNU NPB benchmarks as a single data set, in this context, we train the model on nine benchmarks and test it on the remaining one. Doing so completely excludes the loops of the target benchmark out of a training set, but allows us to get predictions for all benchmark loops, parallelize them if advised so, and test the effectiveness of our assistant. The drawback of this scheme is that it might potentially reduce the accuracy if the nature of loops in the target benchmark dramatically differs from that of loops seen in the training set. Figure 4.3 compares LOOCV accuracy against that of K-Fold CV for all SNU NPB benchmarks, where K-Fold CV is conducted on a data set consisting of loops from a single benchmark only. The comparison proves that lower LOOCV accuracies are attributed to a reduced training data set and not to our ML model.

4.4 Parallelization assistant

The ML-based predictor developed and assessed in the 2 previous sections can have a real practical application. Due to the statistical nature inherent to all machine learn-

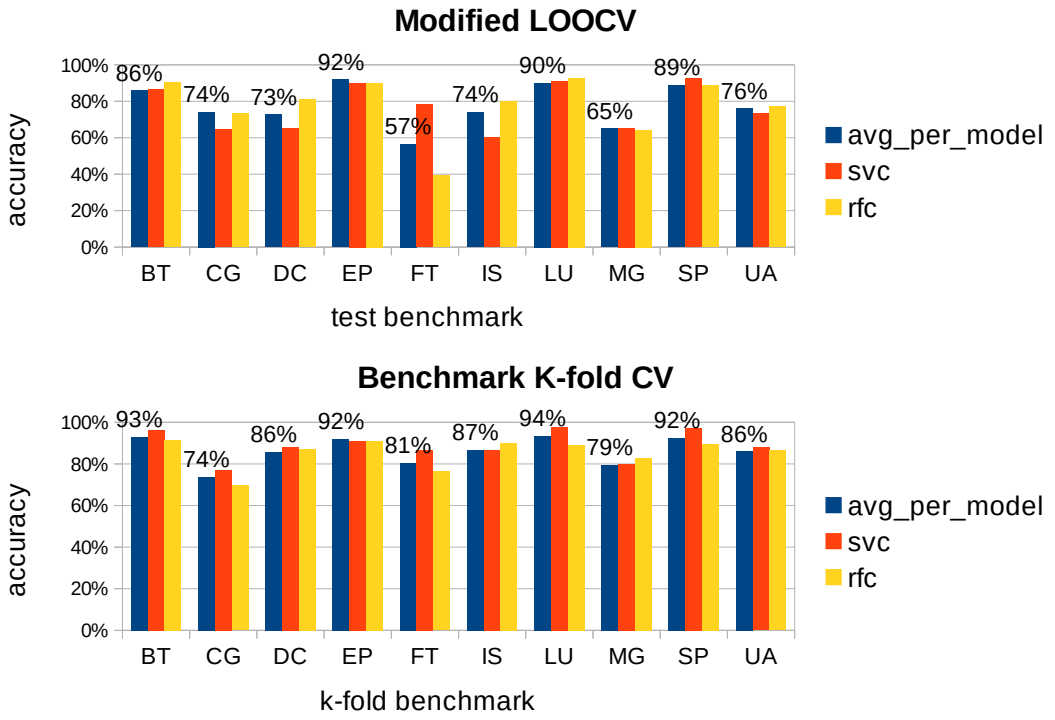

Figure 4.3: Prediction accuracy measured with modified LOOCV and compared against that of K-fold CV for single benchmarks.

ing techniques, it is impossible to eliminate all prediction errors. While false negatives might just miss available parallelization opportunities and lose some performance, false positives can break the program and are the most critical in the context of our ML problem. Given that, we develop a parallelization assistant, which does not seek to replace programmers but aims to assist and increase their productivity. The predictor is the core component of our novel parallelization assistant. The assistant incorporates prediction results and combines them with profiling information. It then produces a ranking of all loops in an application to guide a programmer towards the most beneficial loop candidates for their *manual* parallelization effort.

Loop Ranking. The loop ranking computed by our parallelization assistant combines a loop's contribution to the overall program execution time with its predicted probability of being parallelizable. In particular, we obtain the ranking by applying a shifted sigmoid function to the predicted probability multiplied by the application runtime fraction a loop takes to run as shown in Figure 4.4. The intuition for using this function to combine the two metrics is that it prioritizes parallelizable long-running loops and scales down the weight of non-parallelizable loops irrespective of their contribution to execution time. The effect of this can be seen in Figure 4.5 for the loops of

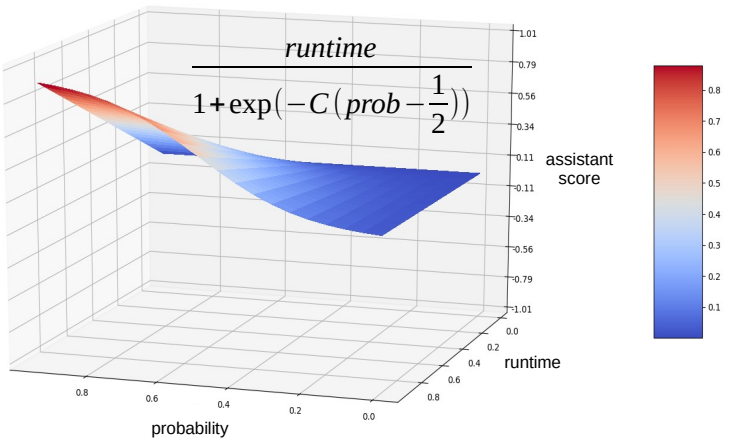

Figure 4.4: For each loop the ranking function combines its contribution to the application's execution time and its predicted probability of being parallelizable.

the FT benchmark. If programmers attempt to parallelize loops in the order prescribed by their execution time, they will inevitably waste their efforts trying to parallelize loops that may be long-running, but offer little or no opportunity for extracting parallelism. Instead, by taking into account predicted parallelizability our ranking directly guides the programmer towards loops that significantly contribute to overall execution time **and** offer a realistic prospect of parallelization.

4.5 Assistant evaluation

4.5.1 Comparison to static analysis

We have compared the generated loop parallelizability classifications of our assistant against that of the ICC Compiler, which due to its use of static analysis is conservative and occasionally misses some parallelization opportunities. The ML approach to parallelization with a human programmer responsible for final code transformation allows our parallelization assistant to be more aggressive than the ICC Compiler. In other words, our model can discover more parallelizable loops.

We have set up an experiment where we apply our ML predictor side-by-side with the ICC Compiler, which has been configured to do the most aggressive parallelization (see Section 4.2.5) on the same machine, that provides all hardware support needed. Both, the assistant and the ICC aim at independently classifying loops as parallelizable

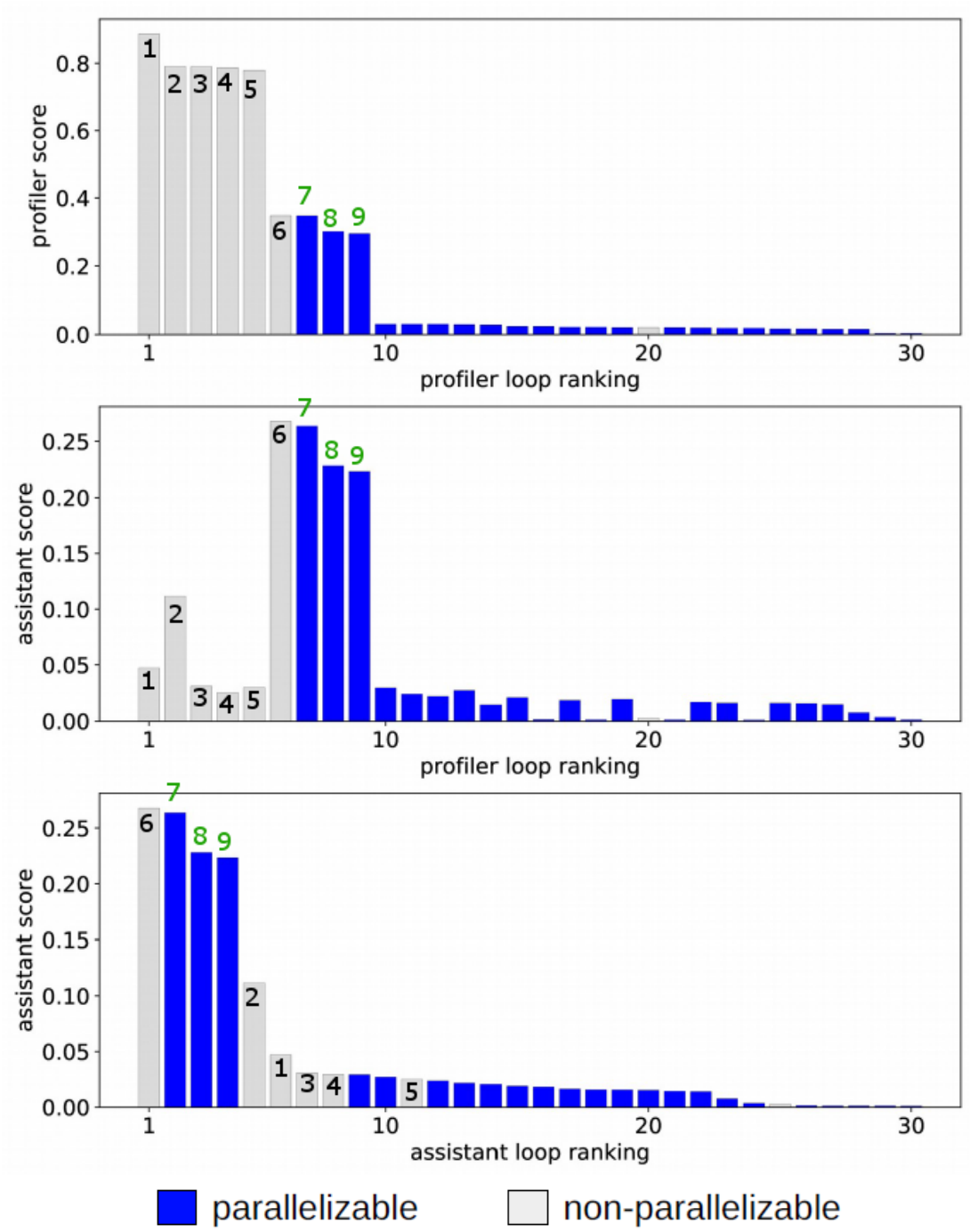

Figure 4.5: Change of loop rankings with the application of the assistant ranking function for the 46 loops of the FT benchmark. Ranking based on loop execution time alone (top figure) results in some high-ranked, but non-parallelizable loops. Combining profiled execution time and parallelizability in a single score (middle figure) results in a ranking that prioritizes parallelizable loops (bottom figure).

Reason	Num	Reason	Num	Reason	Num
missed reduction	18	array privatization	7	conservative analysis	60
unknown iteration number	7	static dependencies	46	too complex	22
non-inlined calls	4	other	4	total	168

Table 4.6: Classification of parallelizable loops rejected for parallelization by the ICC Compiler. The ICC conservativeness is largely explained by the presence of pointers and limitations of alias analysis. Static dependencies are mostly created by indirect array references.

able or not. There is a total of six possible classification combinations that our scheme might produce. Figure 4.6 shows their relative distribution as a pie chart. To calculate the relative frequency of different loop classification combinations illustrated in Figure 4.6 we repeatedly ran K-fold CV on the whole set of SNU NPB loops and sorted the outcomes into separate classification buckets. In the "agreement" cases, which number around 80% of cases the ICC Compiler and ML predictor identically classify truly (non-)parallelizable loops as (non-)parallelizable. This is an expected result, since we used ICC (along with OpenMP annotated loops) to train our ML model. The "missed opportunity" cases, where both ICC Compiler and ML predictor miss parallelizable loops also represent the agreement and are not interesting. The most interesting cases though are those, where ML predictor and the ICC Compiler disagree. While ICC Compiler is conservative and will never classify a non-parallelizible loop as parallelizible, the statistical ML predictor can make a "false positive" error. The rate of false positives in this experimental setting is 8%. That works in the opposite direction as well. The ML predictor can discover truly parallelizible loops, which escape compiler analysis. The rate of such cases is 10%. These cases are classified as "discovery" and have been manually checked in the source code of SNU NPB. The results are summarized in Table 4.6, which reports on the reasons behind ICC conservativeness. False negative mispredictions make ML predictor miss some real parallelization opportunities, but in the fraction of these cases the ICC Compiler can catch them and "shield" the ML predictor.

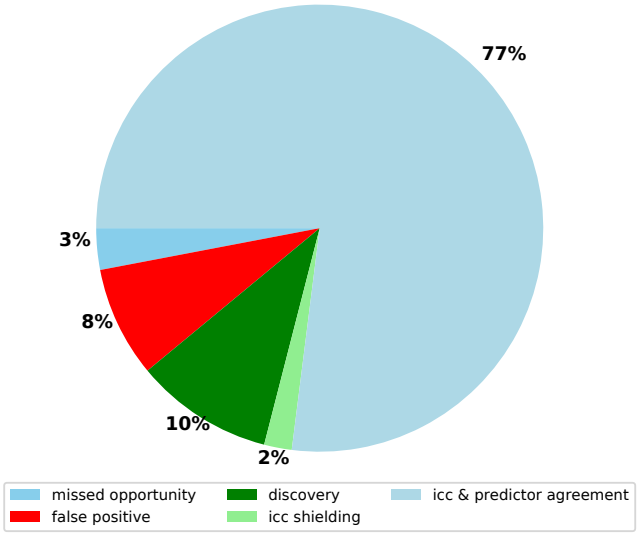

Figure 4.6: Distribution of loop classifications by the ICC Compiler and our predictor.

4.5.2 SNU NAS parallelization

In this section we evaluate the effectiveness of our parallelization assistant. In particular, we are interested in the potential programmer productivity gains delivered by our tool and savings on human expert time. Our study assumes that the human expert starts with a sequential version of the SNU NPB benchmarks. The goal is to parallelize these applications to a performance level matching that of their existing parallel versions. By using our assistant we expect the human expert to consider fewer loops than by using a profiling-based approach, i.e. considering loops in decreasing execution time.

For this experiment we used a desktop Ubuntu 18.04 machine and compiled the benchmarks with the Intel C/C++ Compiler (ICC) 18.0. We compiled the serial versions with *-O3* and *-ipo* flags and explicitly prohibited any vectorization (*-no-vec*) or parallelization (*-no-par*). The OpenMP parallel versions have also been compiled with *-O3* and *-ipo* flags and prohibited automatic parallelization. When we followed the rankings and parallelized benchmarks loop by loop we used OpenMP pragmas and compiled the code with *-O3* and *-ipo* flags as well. Benchmark running times have been measured with the help of UNIX time() utility. To minimize the errors, we ran experiments several times and took the mean average. The machine has 4 Intel Core i5-6500 CPUs with 3.20 GHz frequency and vectorization support up to AVX2. The RAM is 16Gb.

Figure 4.7 summarizes our results as performance convergence curves. For each benchmark the curves plot its execution time (y-axis) as a function of the number of

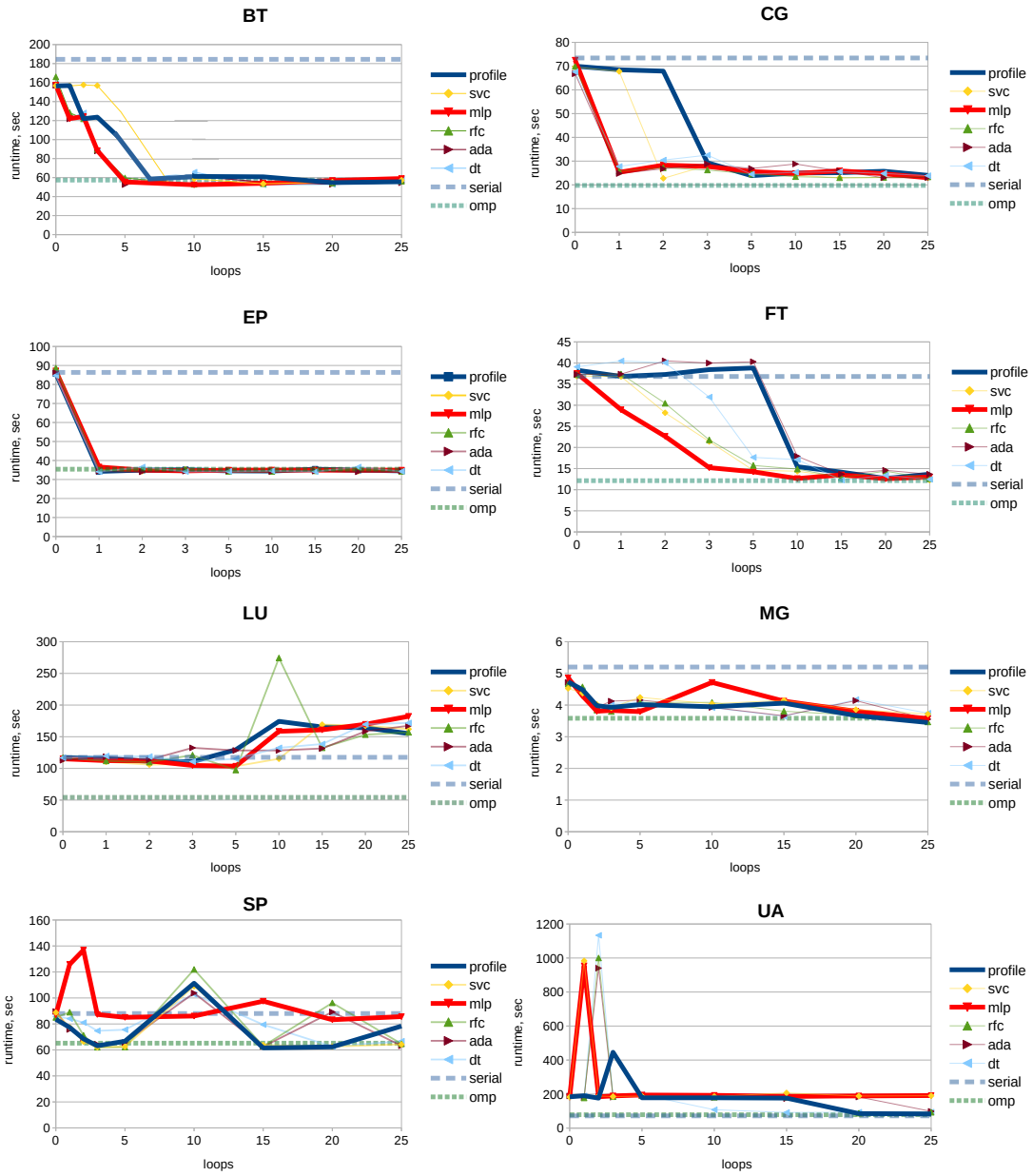

Figure 4.7: From left to right more loops are parallelized for each benchmark. As we parallelize more loops, program execution times improve over the initial sequential performance and reach the performance level of the reference OpenMP implementations. Our ML based parallelization assistant requires the user to parallelize fewer loops than a purely profile-guided approach to reach the maximum parallel speedup.

analyzed and possibly parallelized loops (x-axis). The runtime is bounded within the runtime of the serial execution (top dashed line) and the time of the reference parallel OpenMP version (bottom dashed line). Our goal is to reach the performance of the

reference parallel versions of each program by parallelizing them one loop at a time following the rankings offered by our assistant and the profiler. The neural network based MLP model of our assistant provides the fastest overall performance convergence. While there is some variation depending on the ML model used for the parallelization prediction, in general ML-assisted parallelization outperforms or equals the profile-guided schemes across all benchmarks.

Following the rankings of our assistant in parallelizing the BT, CG and FT benchmarks, we reach their maximum potential performance faster. For BT, maximum parallel performance can be reached after the user has parallelized the first three loops (3061 LOC in Table 4.7) suggested by our assistant, while profile-guided parallelization requires 6 loops (6122 LOC) to be parallelized first before reaching the same performance level. For CG, if we follow the suggestion of our assistant to first parallelize a small loop (6 LOC), we are able to achieve 70% of the maximum potential speedup (see Section 4.1.1). On the other hand, using the profiler ranking requires examining three loops, totalling to 330 LOC, to yield the same performance gains. Moreover, for the SP and UA benchmarks, some of the assistant rankings require the programmer to examine more loops than the profiler. However, the loops proposed by the assistant in the UA benchmark are actually simpler, since they consist of fewer LOC – 508 LOC for MLP and 579 for DT versus the 882 LOC offered by the profiler. By following our assistant's suggestions, a programmer would be required to examine 20% less LOC on average across all models.

In some cases, partial benchmark parallelization might result in a slowdown. In the case of LU, after having parallelized the first 25 loops we do not converge to the best achievable parallel version performance. There are a total of 40 OpenMP pragmas in the benchmark and we need to parallelize all the respective loops to reach the best performance level. In the case of UA, all rankings suggest analyzing a long-running innermost loop first. Its parallelization actually increases running time due to a synchronization barrier being introduced at a wrong program point. It takes 30 loops for the MLP model to achieve the parallel version performance.

Finally, we observe that neither our parallelization assistant nor the profiler reach the performance of the reference OpenMP versions on the DC and IS benchmarks. Manual inspection reveals that these benchmarks have been parallized using OpenMP parallel sections, but do not contain any OpenMP parallel loops. Both our parallelization assistant and the profiler incorrectly suggest to parallelize some of the benchmark loops, though. Table 4.7 summarizes the results of applying our parallelization assis-

Bench.	Bench. Runtime, *sec*			Speedup, *times*		*Loops Number*$_{LOC}$					
	Serial	**OpenMP**	**Critical**	**OpenMP**	**Critical**	**Profile**	**SVC**	**MLP**	**RFC**	**AdaBoost**	**DT**
BT	158.76	57.36	56.57	2.77	2.81	6_{6122}	8_{6392}	4_{4088}	5_{5105}	3_{3061}	5_{5105}
CG	69.38	19.77	25.06	3.51	2.77	3_{330}	2_{118}	1_{6}	1_{6}	1_{6}	1_{6}
DC	698.82	254.29	698.82	2.75	1.00	∞	∞	∞	∞	∞	∞
EP	86.35	35.40	35.07	2.44	2.46	1_{45}	1_{45}	1_{45}	1_{45}	1_{45}	1_{45}
FT	36.81	12.13	14.69	3.03	2.51	9_{338}	4_{187}	3_{140}	4_{187}	9_{338}	5_{193}
IS	4.75	1.35	4.63	3.53	1.03	∞	∞	∞	∞	∞	∞
LU	115.46	55.00	140.53	2.10	0.82	∞	∞	∞	∞	∞	∞
MG	5.20	3.58	3.94	1.45	1.32	3_{43}	3_{43}	3_{43}	3_{43}	3_{43}	3_{43}
SP	86.65	65.19	62.90	1.33	1.38	3_{801}	3_{801}	∞	3_{801}	3_{801}	20_{1257}
UA	71.82	78.56	189.66	0.91	0.38	19_{882}	30_{918}	30_{508}	19_{861}	22_{883}	10_{579}

Table 4.7: SNU NPB benchmark parallelization reports. The left part of the table shows execution times of serial, OpenMP and partially parallelized (critical) versions. The partially parallelized versions have only several critical (top ranked) loops parallelized. The right hand part of the table shows the number of top-ranked loops one needs to parallelize in order to reach the critical performance. The Profile column gives the reference number a profiler requires. The total lines of code (LOC) in the loops are written down as underscript. In most cases, ML based models converge to the critical performance faster than a profiler based approach. There are also cases where a profiler outperforms our assistants.

tant to the SNU NPB suite.

Chapter 5

Computational Frameworks

5.1 Overview

The problem of *parallel software development* is multifaceted. To arrive at the final solution, a programmer has to work on multiple conceptual levels starting from problem decomposition and algorithm choice down to software architecture design, data structure choice, and finer-grained low-level loop transformations. If our loop parallelization assistant (see Chapter 4) aims at reducing the programmer's effort in the task of parallelizing a sequential program, the notion of computational frameworks and the library implementing it could be used in the process of developing a parallel application from the scratch. The solution we propose rids the programmer of the tasks of software architecture design, as well as algorithm and data structure choice by supplying a ready solution for parallelization of certain types of applications. The central motivating ground for the concept of computational frameworks is laid by the *Data-Centric Parallelization (DCP)* problem, limitations of the related work, and the properties of the real world legacy codes.

Motivating ground for Computational Frameworks

The grand problem of Data-Centric Parallelization (DCP) Among all lower level implementation questions, the problem of successful data structure choice stands particularly prominent and important. Listings 5.1 and 5.2 illustrate how easily parallelization can be hampered. Both implementations solve an embarrassingly parallel problem of incrementing every sequence element by one. In the linear array based implementation compiler knows all element addresses statically and can generate parallel code. In the linked list based implementation compiler does not know element

addresses in advance, since the latter will be resolved only dynamically and can generate only sequential code. Moreover, the data structure type is unknown to compiler. By parallelizable and non-parallelizable loops here we mean the loops that can and cannot be parallelized by the state-of-the-art compiler statically.

```
int a[1024];
for (int i=0; i<1024; i++) {
  a[i]=a[i]+1;
}
```

Listing 5.1: Parallelizable loop operating on what is clear to compiler a **linear array**.

```
struct Node* nptr;
for (p=nptr; p!=NULL; p=p->next) {
  p->value+=1;
}
```

Listing 5.2: Non-parallelizable loop operating on what programmer knows is a **linked-list**.

If a programmer had a tool, that could automatically recognize the type and properties of data structures and automatically substitute them with a simpler, parallelizable and more suitable alternatives, that would make the parallelization process a lot easier. Unfortunately, as our state of the art overview discovered, the solution to the problem is not available yet. There are a lot of various techniques and methodologies with their specific limitations and problems.

Data structure and algorithm inseparability An illustrative example in Listings 5.1 and 5.2 is obviously far below the complexity level of the real world code. The work with data structure can be spread all around the code base: allocation and initialization happen in one translation unit, update operations on the data structure are scattered between various functions in multiple other translation units. The exact way an operation works determines the properties (cyclicity, reachability, etc.) of a data structure and ultimately its type. It is crucial to understand how an algorithm actually calls data structure update operations. *This all points to the inseparability of algorithms and data structures: understanding the type of the data structure might require understanding of the algorithm and vise versa; and the data structure substitution might lead to algorithm transformation.* Indeed, as our feasibility studies with the SPEC CPU2006 benchmark suite have shown (Section 2.8.2) let alone automatic techniques, it might take some weeks for even an experienced human software engineer to understand what kind of data structures a benchmark uses and how to optimize it.

Limitations of "The state of the art" work The discovery of higher level entities in programs is definitely not a novel idea. There are various *static* and *dynamic* tech-

niques in the literature. *Shape analysis* is one of the most well-known static techniques, which aims at understanding of various heap-allocated data structures and their properties (node reachability, cyclicity, disjointedness, etc.) at compile time [18][25][26]). *Shape analysis techniques give a very rough and conservative approximation (a tree instead of a linked list), work with high error rates (DAG instead of graph with cycles) and are provably undecidable [21].* One of the more recent static techniques is based on pattern matching on the code intermediate representation (IR) level and aims at recognition of various computational idioms (such as reductions, stencils, sparse and dense linear algebra computations, etc.)[68][73][74]. Computational idioms are specified in a constraint-based domain specific programming language CAnDL. *The technique allows for a rapid prototyping of new compiler optimizations based on pattern recognition and its substitution with an optimized versions of matched idioms, but it is limited for a relatively simple computational idioms and entities.* All static methods are limited in their program view to a single compilation unit. The challenge of data structure recognition requires a much broader view. Dynamic techniques come the closest to the solution of the DCP problem. There are numerous works available [64][70]. These techniques are based on program instrumentation and construction of various dynamic memory allocation graphs. The idea is that these graphs along with their update operations can reflect the actual shape of the data structure. Probably, the most promising and the best performing of all is the work of a Data-structure Detection Tool (DDT) [41]. The DDT tool can successfully recognise data structures in most of the standard libraries, such as STL, Apache (STDCXX), Borland (STLport), GLib, Trimaran achieving almost perfect recognition accuracy. Moreover, the technique has been able to recognise linked lists in Em3d and Bh Olden benchmarks. But, it is still far from solving the problem for an arbitrary real world code.

The ongoing trend to higher abstraction levels For decades there has been an ongoing trend in the process of software engineering to move up in the levels of abstraction from bare hardware to higher level concepts closer to human reasoning and understanding. We have moved from assembly languages to languages like Fortran and C, followed by the development of object-oriented languages and a supplement of imperative programming languages with functional programming concepts. All these steps increase programmers productivity, improve program structuredness and modularity and move the process of software design closer to a human level. The trend of moving to higher abstraction levels is not only true for software engineering in general, but for parallel software engineering in particular. For example, standards like POSIX, OpenMP and

MPI aim at abstracting a programmer from various hardware and operating system details and work on the level of platform agnostic parallel programming models. Parallel algorithmic skeletons [54] (see Section 2.6) move software parallelization process even higher and allow a programmer to specify a computation on the algorithmic level with various concepts like *map*, *stencil*, *divide and conquer*, etc.

A higher level solution

The task of separating data structures from algorithms for an arbitrary real world code is extremely challenging. Moreover, for some applications and benchmarks it does not seem meaningful either. For example, the suite of Olden benchmarks is much simpler, than SPEC CPU2006, but also exhibits the same inseparability problem. For many of Olden benchmarks the data structures and algorithms are blended, but the union they form can be framed into an elegant higher level entity, that can later be used to parallelize the benchmarks in a nice and structured way.

In this thesis we propose a novel notion of *computational frameworks*, which exploit this possibility and help a programmer with a coarse-grained parallelization tasks, such as software design, algorithm and data structure choice on applications, where computations can be expressed with our computational frameworks. We describe the concept and the major frameworks in Section 5.3. We provide a prototype C++ template library implementing the notion [82] (see Section 5.3.5). We prototyped the library on the suite of Olden benchmarks, which inspired and shaped the concept. The parallel library version consistently outperforms the sequential version hitting 5-6x speedups on the major benchmarks (see Section 5.4).

5.1.1 Contributions of our Computational Frameworks

▷ We propose a novel notion of *computational frameworks*, which are higher level entities that embody both data structures and algorithms; and

▷ report on a prototype C++ template library [82] implementing the notion in a modern, convenient, parallel and an easy to use way;

▷ We express the computations of some Olden benchmarks in terms of our computational frameworks and rewrite their original sequential legacy C versions with the help of our library in a modern, better structured, portable and crucially parallel way (see Section 5.3);

▷ We demonstrate the potential of the notion and the performance of the prototype library on the suite of Olden benchmarks (see Section 5.4), achieving consistent parallel speedups of 5-6x on the major benchmarks;

▷ Finally, we propose an idea of alternative software parallelization approach based on our computational frameworks as a future work.

5.2 Usage example

To get an initial feeling for what it is like to use our computational frameworks from a user perspective, let's consider a motivating example. A full discussion of the concept follows in Section 5.3. Suppose we want to calculate a well-known functional programming concept of the *right fold*, which also updates its elements with corresponding folded values, thus leaving some *side effects*.

We can code that simple computation using the C++ Standard Template Library (STL). Listing 5.3 shows the code implementing the task with an STL $list <>$ class template. First, we construct a list with 5 elements and initialize them to the value of 1. Then we loop through the list updating its elements and return the final right folded value. The task is small and the code is concise, but it is still possible to see its drawbacks. The code does not clearly separate concerns: list traversal, result computation and list transformation all happen in the same place and are mixed up together. For a better structuredness and comprehensibility it would make sense to put these different pieces of functionality into separate places. Alternatively, we could use STL's $std :: accumulate()$ function template. The latter provides a programmer with a more concise and abstract interface, but does not allow any side effects. We would still need to write a separate chunk of code aimed at implementing an update of the list elements.

Listing 5.4 shows an alternative implementation of the right fold using our **Fold** computational framework. The main computation here is expressed with a single line of code and a reader familiar with the concept of fold will comprehend the purpose of the code instantly. The fold defines the backbone of the computation, which is hidden from a user, while the definition of the custom part is left for a user to complete and is passed into a higher order functional style interface $Fold < Elem >:: compute()$ as a function object $ComputeFunc$, which has been derived from a Fold-specific base class $Fold < Elem >:: ComputeFunction < int >$. The latter is a template with the main computational result specified as a parameter. The base class frames the interface. Op-

erator $Fold < Elem >:: ComputeFunction < int >:: operator()(Elem\&, int)$ takes an element of the Fold as an argument and combines it with the value folded so far in a user defined way. The user is supposed to pass the result further and has a freedom to update the element, thus leaving some side effects. The code in Listing 5.4 has a clear structure, but might feel a bit heavy for such a simple computation, but when a computational task is significant the advantages of the shown code design will become clear.

```
#include <list>
using namespace std;

int main() {
    list<int> lst(5, 1);
    // lst = [ 1 <- 1 <- 1 <- 1 <- 1 ]

    int result = 0;
    for (auto it = lst.rbegin(); it != lst.rend(); it++) {
        *it += result;
        result = *it;
    }
    // lst = [ 5 <- 4 <- 3 <- 2 <- 1 ]
    // result = 5

    return 0;
}
```

Listing 5.3: Right fold computation using standard STL list class template

```
#include "Fold.h"
using namespace abstract;

class Elem : public Fold<Elem>::Element {
    public:
        void grow() override { value = 1; }
        int value;
};

class ComputeFunc : public Fold<Elem>::ComputeFunction<int> {
    int operator()(Elem& elem, int fold) override {
        elem.value += fold;
        return elem.value;
    }
```

```
};

int main() {
    int fold_depth = 5;
    Fold<Elem> fold(fold_depth);
    // fold = [ 1 <- 1 <- 1 <- 1 <- 1 ]

    int result;
    ComputeFunc comp_func;
    result = fold.template compute<int>(comp_func);
    // fold = [ 5 <- 4 <- 3 <- 2 <- 1 ]
    // result = 5

    return 0;
}
```

Listing 5.4: Right fold computation using our Fold computational framework

5.3 Computational Frameworks

In this section we describe the general concept of **computational framework** along with all frameworks we propose and implement in the prototype library. As our frameworks have been inspired by computations found in the Olden benchmark suite, it would be helpful to review the Section 2.8.3 describing the key benchmarks.

5.3.1 The concept

As we have already mentioned the concept of *computational framework* has been inspired by the current problems in the software parallelization field along with computational patterns we have observed in the suite of Olden benchmarks. Very often programs are written with sub optimal from the point of software parallelization data structures. It might be hard to substitute them with parallelizable alternatives as the former might be closely entangled with algorithms the program is based on. Hence:

▷ Computational frameworks grow on the problem of data structure and algorithm inseparability;

We might keep the two together, but still tackle the problem at a higher level. We call the higher level entity we operate with a computational framework. Computational

emphasizes the algorithmic component, while framework hints towards the underlying data structure.

▷ Computational frameworks combine algorithms and data structures to form a higher level entity;

Moreover, there are some specific problems, that could be tackled more effectively with specialized constructs. Imagine some task of scientific simulation. The scientific computation might be abundant with various higher level algorithmic constructs like maps, reductions, folds, stencils, etc. These concepts are characteristic of functional programming. The latter often forbids any mutable states. At the same time simulations often keep a significant state being updated and accumulated with every step. The presence of state is characteristic to an imperative programming. Here one can see a contradiction and hence the gap to fill:

▷ Computational frameworks fill the gap between imperative and functional programming paradigms;

With all the things said above it is human programmers, who are going to use the concept in the end. Hence, it must be human friendly and convenient. At the same time the code must be modern and effective. We designed an object-oriented library with a functional style interface consisting of higher order functions. While the algorithmic component of the given framework is immutable there is a custom user defined functionality to apply to the framework. That functionality comes as a function object through the framework's interface in accordance with a command and template method (see Section 2.5.2) software design patterns.

▷ Computational frameworks provide a modern and convenient interface with elements of both functional, as well as object-oriented programming; and

▷ Computational frameworks embody the best ideas of various software design patterns.

When the software architecture has been designed, the exact efficient algorithm has been chosen along with all the most optimal and suitable data structures supporting its operation, a programmer can move onto the task of software parallelization. Here:

▷ Computational frameworks implement an effective and portable parallelization under the hood.

To conclude:

▷ Computational frameworks improve program **structuredness**, **modularity**, **separation of concerns** and hide possible **program parallelization** behind the **modern and convenient** user interface.

5.3.2 Fractal

The **Fractal** computational framework is the key framework in our work. It has been inspired by the theoretical work on tree reductions [65] and 3 Olden benchmarks (*health*, *treeadd* and *perimeter*) with a very similar computational pattern. Although, the work on tree reductions has laid the theoretical foundation, but it did not provide a practical implementation. While there are numerous computations, processes and examples, which can be framed into the fractal, for the simplest and the most illustrative example let's consider an n-ary tree data structure being processed as follows.

Fractal's growth. First, we grow the tree from the root down to its leaves. A node takes a seed to grow, grows and spawns the next set of seeds for all its children nodes. The latter continue the process of growth. In the most general case the growth can happen asymmetrically, i.e it can stop on some paths down the tree, when some user specified condition is met for some nodes. In other words, the tree becomes incomplete with some leaves growing deeper than others and some nodes not having some children. We saw that pattern in the *perimeter* benchmark (see Section 2.8.3), when squares fell completely outside or inside the ring, they stopped to split further. Benchmarks *health* and *treeadd* grow complete trees and do not specify any stop conditions. While the algorithmic pattern of growth is fixed, the node's growth itself along with the node's data are custom and user defined parts. The same is true of the type of seeds and seed spawning procedures. Moreover, the growth of the tree along its various branches happens independently and can be parallelized.

Fractal's processing. When the tree is grown, we can start to process it. On that stage we go in the opposite direction from the bottom up by starting to process the leaves of the tree first. For every node we do some custom computation along with updating its custom data and pass the computation's result up to its parent. Before we process any node we need to process all its children first. The processing of different children happens independently and can safely be parallelized. When the tree has been processed we obtain the final result from its root along with all the side effects left in the nodes of the tree. The user defines these side effects per node in the custom

processing function along with defining the exact computation from children up to the parent. We can repeat the process numerous times. This is the way some simulations work. The side effects are going to accumulate and make the nodes heavier. Figure 5.1

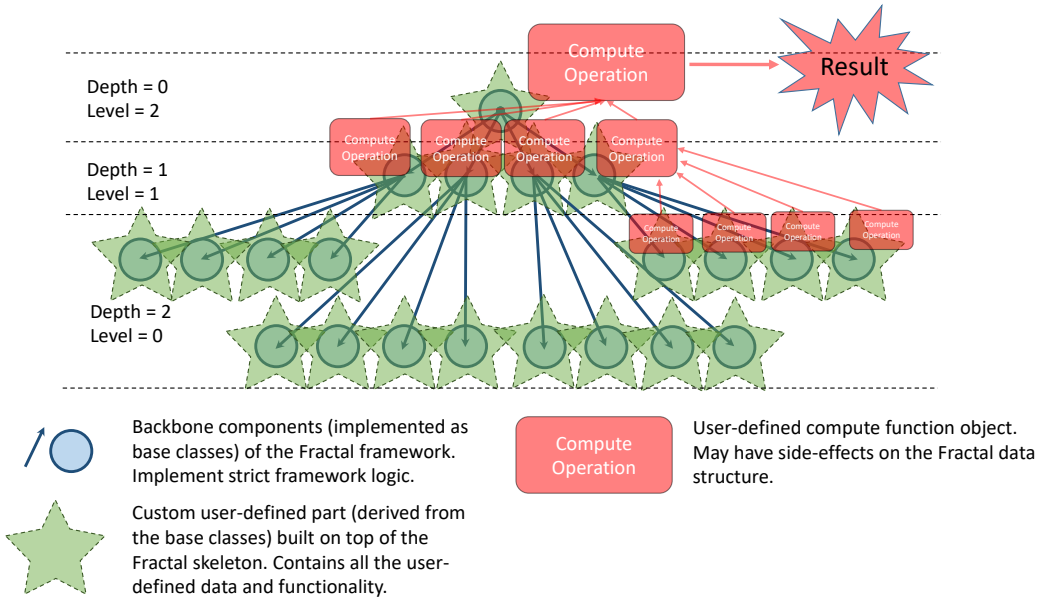

Figure 5.1: Our object-oriented design of the Fractal computational framework.

illustrates the pattern described above, as well as proposes an object-oriented design of the fractal concept. The backbone components (circles and arrows) of the tree are immutable and keep the main computational and growth patterns. These components can be implemented as base classes. Along with the backbone logic implementation these base classes provide a customizable interface to be overriden by derived classes (stars). Derived classes specify custom data contained in the tree nodes along with the exact growth and processing procedures, which can touch and alter the data. Given the fixed custom data in the tree nodes, computations operating on the data may still vary. We customize computations as function objects (red rectangles) being passed into higher order functional interfaces of the fractal. In other words, our fractal provides functional style interfaces above its object-oriented structure with the main backbone logic hidden under the hood. The backbone logic can be implemented sequentially or be parallelized in a multiple ways. In all its generality the fractal is a pattern, which can be characterized with self-similarity, repeatedness, structuredness, inherent parallelizability and the exact numeric values such as its depth and arity.

5.3.3 Fold

The **Fold** computational framework has been inspired by the computation done in the *power* benchmark (see Section 2.8.3). The fold is not a new concept and has found a wide application in many functional languages. The C++ language provides *std::accumulate()* function template as a component of its Standard Template Library (STL), which performs a functional fold over a given data structure, but contrary to our computational framework it does not allow any side-effects and modifications to the elements of the data structure must be coded separately. Our C++ *Fold* class template provides an alternative interface to a user with an extensible customization space and clear separation of concerns.

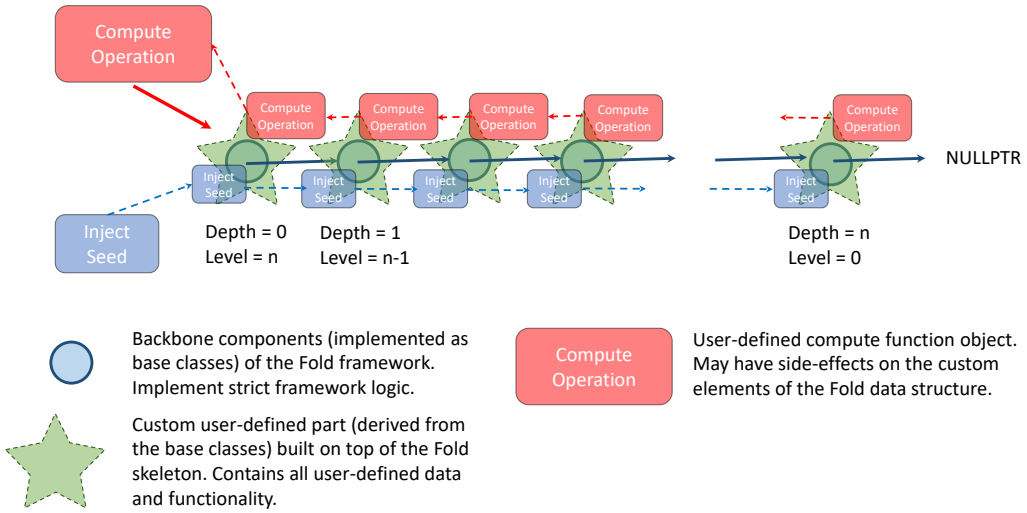

Figure 5.2: An object-oriented design of the Fold computational framework.

Figure 5.2 illustrates the framework. One can think of a Fold as a set of elements arranged into a linked-list. We grow the list to the specified depth given a seed value. Then we may inject some data into the head of the list and propagate the data along with its custom changes to the last element at the tail of the list. All propagation modifications are user-defined. The injection of the data might happen as needed (before every fold iteration for example). Once every element of the list is ready with its injected data planted, the computation starts at the tail element and passes computed values back to previous elements of the chain. The computation may leave some side effects on the elements of the fold, which can accumulate with fold repetitions. The

pattern is not parallelizable, but defines a strict order and helps to structure the code to separate various concerns. The object-oriented design of the fold framework is coherent to that of the fractal. Base classes form the skeleton logic and define the interface. All customization happens through overriding inherited base class interface methods withing definitions of derived classes. Custom compute operation is a function object going into functional interfaces of higher order *Fold* class methods.

5.3.4 Reduce

Strictly speaking, **Reduce** is not a computational framework, but an algorithmic skeleton (see Section 2.7.1). Although it is implemented with exactly the same design and interface, what illustrates compatibility between the two closely related concepts. The difference between our framework and *std::reduce()* from C++ Standard Template Library (STL) is the possibility of having side effects and an alternative user customization interface. All general remarks made regarding **Fold** and **Fractal** frameworks also apply to the **Reduce** framework, although the specific details might differ. For example, the reduce framework takes a function object with two overloaded and virtual *operator()* methods. One specifies how to reduce the value from a single element (possibly changing the element in the process) and the other defines the way of combining all the reduced values into the final return value. Our framework implements sequential as well as parallel **Reduce** versions.

5.3.5 Frameworks library design and implementation

Despite a number of framework specific differences, all our frameworks stick to the same user interface and design. The design and implementation of computational frameworks library have been done along the curved and iterative path running into numerous dead ends and doing redesign efforts over and over before we managed to get the final library version [82]. The major questions raised during the design process of the library were the trade-off between employing static vs. dynamic polymorphism, interoperability of different frameworks (how do we handle the reduction of folds of folds of reductions like in the case of the *power* benchmark, for instance?), as well as the overall library coherence questions (how to design a common interface for patterns as different as fold and fractal?). But the final design goals have always been clear and follow below.

Modern C++ The implementation of the library is based on the Standard Template

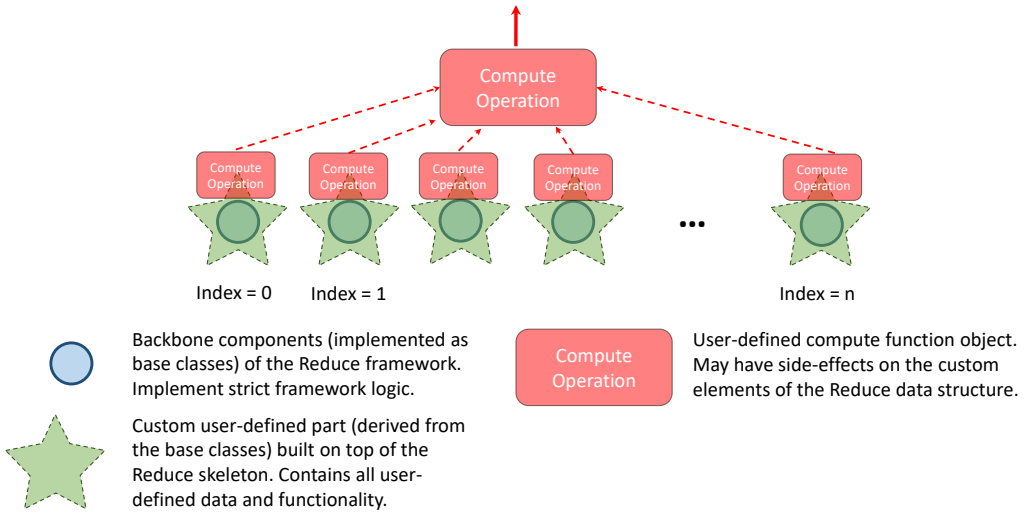

Figure 5.3: An object-oriented design of the Reduce algorithmic skeleton. The design is identical to that of computational frameworks. The two concepts are inter-operable.

Library (STL) data structures, uses move semantics and unique pointers to achieve efficiency and smart memory management. For parallelization library uses OpenMP standard, thus making source code portable. The library is composed of a set of header files with class templates, which are supposed to be included into the user application.

Convenience We designed the library in an object-oriented fashion with a functional look of its user interfaces and the portability in mind.

Coherence Despite variations in the prescribed behaviour, all computational frameworks should stick to the same user interface as well as internal design. The coherence would improve inter-operability of different frameworks (say composing a reduction of folds), improve library usability, as well as ease its maintenance and extension. Among less intuitive things, a more general design that handles folds, reductions, fractals, etc. along with all the applications using them altogether will be of the higher quality overall.

Sound design We strove to use the best software design practices. The user side of the library has been inspired by the LLVM Pass Framework [1] and alike it the library uses the *Curiously Recurring Template Pattern (CRTP)* to decrease the number of template parameters and avoid some of the dynamic polymorphism overheads. The concept of computational frameworks fits exactly into the well-known *algorithm template* pattern and the functional user side is implemented with function objects and higher order

template methods following the *command* pattern.

We used 4 Olden benchmarks as an inspiration and guide. Benchmarks *health*, *perimeter* and *treeadd* defined our **Fractal** computational framework. The *power* benchmark shaped **Fold** and **Reduce** frameworks. Moreover, striving for coherence and common interface, the designs of different frameworks had a profound effect on each other.

Listing 5.5 shows the essential parts of the final framework design. Initially, custom framework element growth methods such as $Element :: grow()$ and $Element :: growth\_stop\_condition()$ were specified as separate $Framework <>$ template parameters, but were moved to become virtual functions of the base $Element$ class as a trade-off between static vs. dynamic polymorphism. The template method $Framework <>:: compute < ComputeType > ()$ is a higher order functional interface, which takes a function object of framework specific type $Framework <>:: ComputeFunction <>$ and applies it to the framework along with computing the main result of the $ComputeType$ type. Listing 5.5 shows the one for **Reduction** class. A user is supposed to override two overloaded operators(), which specify a reduction from a single element complemented with the final combining operation. $Framework <>:: grow()$ method defines the backbone data structure and representation of the framework and makes calls to custom user defined functions controlling the growth process.

```
template <typename ElemType, typename SeedType>
class Framework {
    public:
        class Element {
            // user-exposed customization iface
            virtual void grow(SeedType) = 0;
            virtual bool growth_stop_condition() { return false; }
        };
        template <typename ComputeType>
        class ComputeFunction {
            public:
            // framework specific application function API
            virtual ComputeType operator()(ElemType& elem) = 0;
            virtual ComputeType operator()(const std::vector<
    ComputeType>&) = 0;
        };
        void grow(size_t size, SeedType seed) {
            // organise framework elements
            // into a data structure
```

```
            ... = new ElemType();
        }
        template<typename ComputeType>
        ComputeType compute(ComputeFunction<ComputeType>& apply_func
    );

    private:
        // framework data structure organisation
        // (list, tree, array, etc.)
};
```

Listing 5.5: Computational framework class template skeleton

5.4 Library Deployment

5.4.1 Source Lines Of Code (SLOC) metric comparison

Although the Source Lines Of Code (SLOC) metric does not always correctly reflect program properties (such as comprehensibility, maintainability, etc.), it is still interesting to compare the original legacy C implementation of Olden benchmarks against the implementation based on our computational frameworks in terms of this metric.

Here we need to make some comments. First, the comparison is not completely fair, since the original pointer-based implementation of Olden benchmarks is sequential (only *power* benchmark contains parallelizing OpenMP pragmas). Whereas our implementation can be made sequential or parallel with just one line of code (like *tree.set_impl_type(Fractal_t::ImplType::parallel)*) and everything else happens ”under the hood”. It is also important to note, that we are comparing legacy C vs. C++ implementations. Programs written in C++ tend to be slightly more verbose in terms of SLOC with their class definitions, constructors, destructors, extra keywords, etc. Second, as with any third-party library, before we can use it, we need to perform a setup, which often contains a lot of ”boilerplate” code. In the case of computational frameworks, the setup implies the derivation of custom classes from the base classes of our library, instantiating templates, and some other extra syntactical constructions. The business logic of the application can be split into two phases: computational framework *build* and *compute*.

Table 5.1 presents the results. For all 4 benchmarks, the setup size is larger in terms of SLOC for the implementation based on our computational frameworks, but after the

stage	power		health		perimeter		treeadd	
	original	abstract	original	abstract	original	abstract	original	abstract
setup	80	170	55	65	15	35	10	30
build	60	50	50	30	50	60	15	1
compute	380	460	250	230	190	200	60	40

Table 5.1: Implementation SLOC size comparison: original legacy C version (*original*) vs. C++ computational frameworks based one (*abstract*).

setup is done we can see some improvement on the *health* and *treeadd* benchmarks. Parallel C++ version of the benchmarks takes less SLOC than a legacy sequential one. The *power* benchmark demonstrates the opposite tendency. The computation of the benchmark is composed of the reduction of folds of folds of reductions. Moreover, every fold contains two components: left inject and right fold. These joint points require some additional setting, initialization, and passing of parameters. While this improves the separation of concerns and program structuredness, it bloats the code as a side effect. The *perimeter* benchmark does not show significant differences.

5.4.2 Performance study of the library

To assess the potential and utility of the concept of computational frameworks, we implemented a prototype library and conducted its thorough performance examination on the subset of Olden benchmarks. We used 4 benchmarks we are interested in. Benchmarks *health*, *treeadd* and *perimeter* fit into our **Fractal** framework and stress it from different angles. We used the *power* benchmark to test the practical operation of our **Fold** and **Reduce** frameworks. We expressed computations in these benchmarks through applicable computational frameworks and have rewritten the original sequential legacy C implementations of the benchmarks with our prototype library in a rejuvenated, structured, well-designed and crucially parallel way. On the opposite side, these benchmarks inspired the concept of computational frameworks and shaped the design of the library. *All our benchmarks have been compiled with GCC 10.1 and -O3 optimization sets. Benchmark running times have been measured with the help of UNIX* ***time()*** *on a powerful compute cluster running Ubuntu 20.04 (focal) and having 64 AMD EPYC 7302 3303 MHz 16-core processors with a total of 0.5 Tb of RAM.*

Performance plots below (Figures 5.5, 5.6, 5.7, 5.8) show the strengths of the li-

brary, as well as its equally important weaknesses. The latter characterize applicability and limitations of the library, rather than the problems of the idea. Although *perimeter* benchmark still demonstrates the potential it also highlights the prototype research nature of the library and shows where it needs an optimization effort. The latter presents a matter of software engineering and not that of a research.

The implementation can be easily configured to be sequential or parallel, thus making it easy to conduct measurement experiments. Moreover, the Fractal framework has 2 implementations under the hood. In the most general case the Fractal is based on N-ary unbalanced tree. In this case we allocate it dynamically and every node has an array of pointers to its children. But in the case of a perfect (all leaves are at the same depth and every non-leaf node has all children present) tree, we can allocate all nodes linearly in an array-based heap manner. In the latter case parallelization happens per level, spawning the number of threads equal to the number of nodes at the given depth (provided that enough CPU cores are available). In the general case though we do not know where the growth of the fractal is going to stop and cannot index nodes on the array. Parallelization in the case of unbalanced fractal happens per child: spawning the number of threads equal to the number of children (again, provided that enough CPU cores are available). Figure 5.4 illustrates two implementation strategies and performance plots below compare them. Listing 5.6 illustrates ”under-the-hood” parallelization of our fractal computational framework in the most general case of unbalanced tree. Parallelization happens only at the topmost level to limit the number of threads spawned.

```
template <typename ElemType, typename SeedType, int Arity>
template <typename ComputeType>
ComputeType Fractal<ElemType,SeedType,Arity>::Element::compute(
    ComputeFunction<ComputeType>& compute_func) {

     std::vector<ComputeType> ret_vals;

     if (!children.empty() &&
         (this->info.level-1 > 0) )
     {
         if (info.depth < 2) {
             ComputeType tmp[Arity];
             int threads_count = Arity;
             #pragma omp parallel num_threads(threads_count) shared(
    tmp)
```

```
            {
                #pragma omp for schedule(static)
                for (int i = 0; i < Arity; i++) {
                    tmp[i] = children[i]->template compute<
    ComputeType>(compute_func);
                }
            }

            for (int i = 0; i < Arity; i++) {
                ret_vals.push_back(tmp[i]);
            }
        } else {
            for (int i = 0; i < Arity; i++) {
                ret_vals.push_back(children[i]->template compute<
    ComputeType>(compute_func));
            }
        }
    }

    return compute_func(*(static_cast<ElemType*>(this)), ret_vals);
}
```

Listing 5.6: The ”under-the-hood” parallelization of unbalanced Fractal computational framework

Figure 5.5 demonstrates performance of various versions of the *health* benchmark. The benchmark was described in Section 2.8.3. The benchmark grows a tree of hospitals and performs a simulation of the Columbian healthcare system step by step. The number of simulation steps done is plotted on the horizontal axis. The time a simulation took is plotted on a vertical axis. Nodes of the tree, which represent hospitals grow various lists of patients. As simulation goes the lists grow and the nodes become heavier. In other words, the state of the benchmark becomes bigger. The latter has an accumulating effect and contributes to the workload. We can see this phenomenon reflected in an exponential time growth of the sequential version. Parallel versions diminish this time by tackling the task with several threads. This results into 5-6x speedups of the parallel versions relative to sequential ones. This benchmark is an ideal one to tackle with our fractal framework. And indeed, as Figure 5.5 shows, the thick lines representing the real time (wall clock time) indicate that a parallel versions consistently outperform sequential versions. The sequential versions of our library

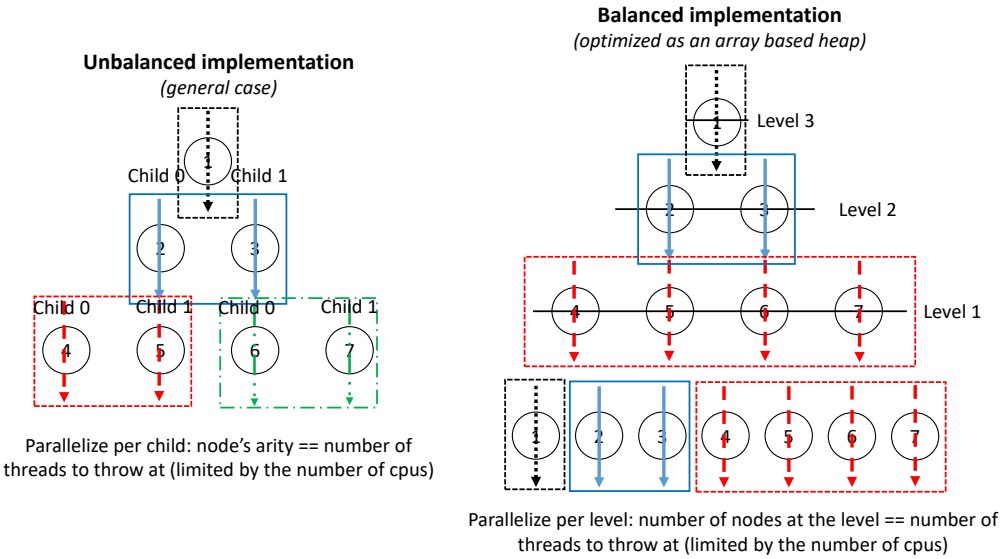

Figure 5.4: *Balanced* vs. *Unbalanced* implementations of the fractal framework. Unbalanced implementation handles arbitrary trees and thus is based on pointers. Balanced implementation optimizes the case of a perfect tree and can be implemented with an array.

perform roughly as well as the original legacy C implementation (thin lines vs. a thick *original* line). The latter shows that our library does not introduce any overheads on a benchmark with a significant workload. Dotted lines show the CPU time and illustrate how much of the CPU time has been used by several computational threads of the application. This measure can be used to judge on the aggressiveness of parallelization.

Figure 5.6 illustrates the results for the *power* benchmark. The benchmark performs a workload of scientific computations. The pattern is basically a reduction of folds of folds of reductions. So, the benchmark tests our **Fold** and **Reduce** frameworks. We can vary the width of reductions as well as the depth of folds to get various amounts of workload. The benchmark runs repeatedly until the computation result falls withing the set epsilon error. Figure 5.6 illustrates how the running time of the benchmark scales with an increasing top level reduction width. The latter increases the workload and one can see the growing running time of the sequential original legacy C version. The sequential version based on our library runs roughly as well as the original one, while the parallel version consistently outperforms both sequential versions. Dotted line again represents the CPU time and not the real time. CPU time approximately reflects how many threads were used to run the benchmark. As a validation, one can

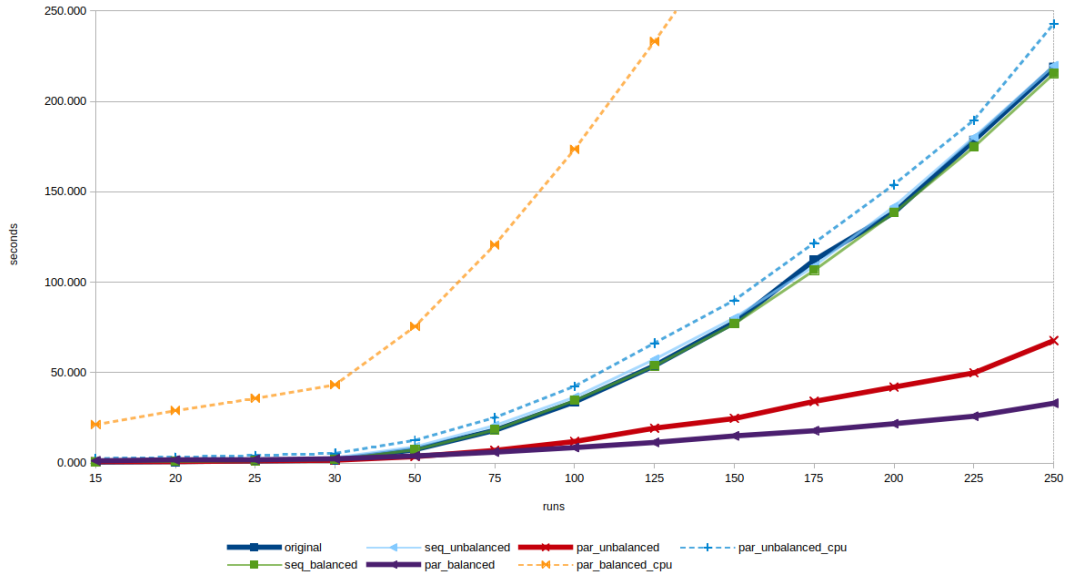

Figure 5.5: Library performance on the *health* benchmark. The legacy C implementation is represented by a thick *original* line. Implementations of the benchmark based on our library are represented by 2 thick (*parallel balanced* and *unbalanced*) and 2 thin (*sequential balanced* and *unbalanced*) lines. Dotted lines represent the CPU time, which is not the real time and has a secondary importance. CPU time reflects the computational workload as we throw several threads at a task and can be used to judge on parallelization aggressiveness.

see that CPU time divided by parallel wall clock time roughly corresponds to the reduction width. Behaviour of the *power* benchmark does not change significantly, when we vary reduction widths or fold depths. We do not include the other experiments we ran, as they do not change the picture.

The *health* and the *power* benchmarks show the strengths of our library. The *treeadd* and *perimeter* benchmarks highlight its weaknesses. The weaknesses stress the research prototype nature of the library and show places for further optimization effort. Figure 5.7 shows the behaviour of the *treeadd* benchmark. This benchmark grows a tree with values at its nodes. Then it runs iteratively computing the reduction over the tree and updating node values. The computation is very lighweight. Basically it is just one addition per node processing. The state is minimal and does not accumulate as we run the benchmark over and over. One can see a roughly linear running time of the original legacy C version. Overheads of our sequential library versions are striking for such a small benchmark. Although the asymptotic complexity of all shown versions is the same, various bookkeeping overheads of the library diminish performance relative

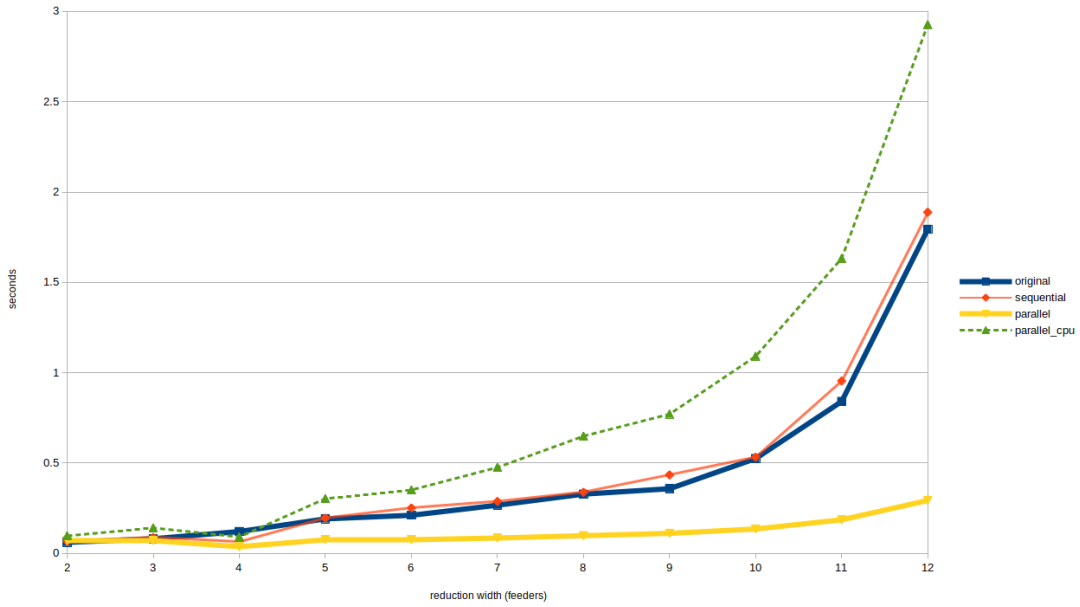

Figure 5.6: Library performance on the *power* benchmark. We vary the top level (feeders) reduction width to scale the workload.

to the original version with just a single addition per node as the workload. These overheads can be optimized away with some engineering effort, but present in the research prototype library. The more threads we throw the bigger our overheads become. One can see it looking at the running time of balanced and unbalanced versions. It is worthless to tackle such a small benchmark with our library.

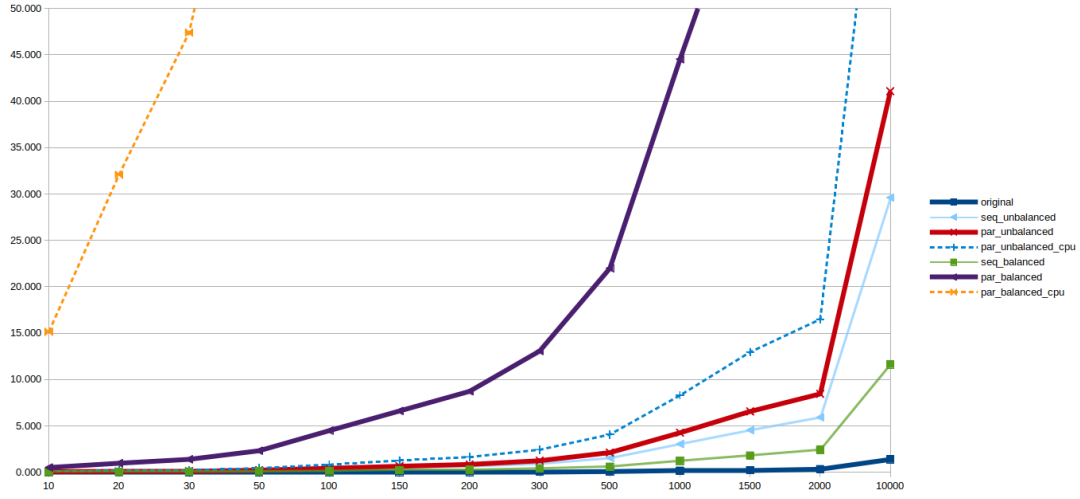

Figure 5.7: Library performance on the *treeadd* benchmark.

The *perimeter* benchmark shows a different picture, but still highlights the current weaknesses of the library implementation. Figure 5.8 shows the bar chart. The bench-

mark does not run an iterative simulation as the 3 other benchmarks do - it is a single perimeter computation. Horizontal axis plots the depth of the tree underlying perimeter computation, a workload size in other words. As usual, the vertical axis plots the time it took to run the computation. As the benchmark is based on an unbalanced tree version we cannot run it with a balanced implementation. Here we have only the original sequential legacy C implementation, the sequential implementation based on our library and its parallel counterpart. One can see that the benchmark also highlights the current implementation problems: original does better than sequential. At the same time the benchmark has a potential: there is a good parallel speedup when we compare the sequential versus parallel library based versions.

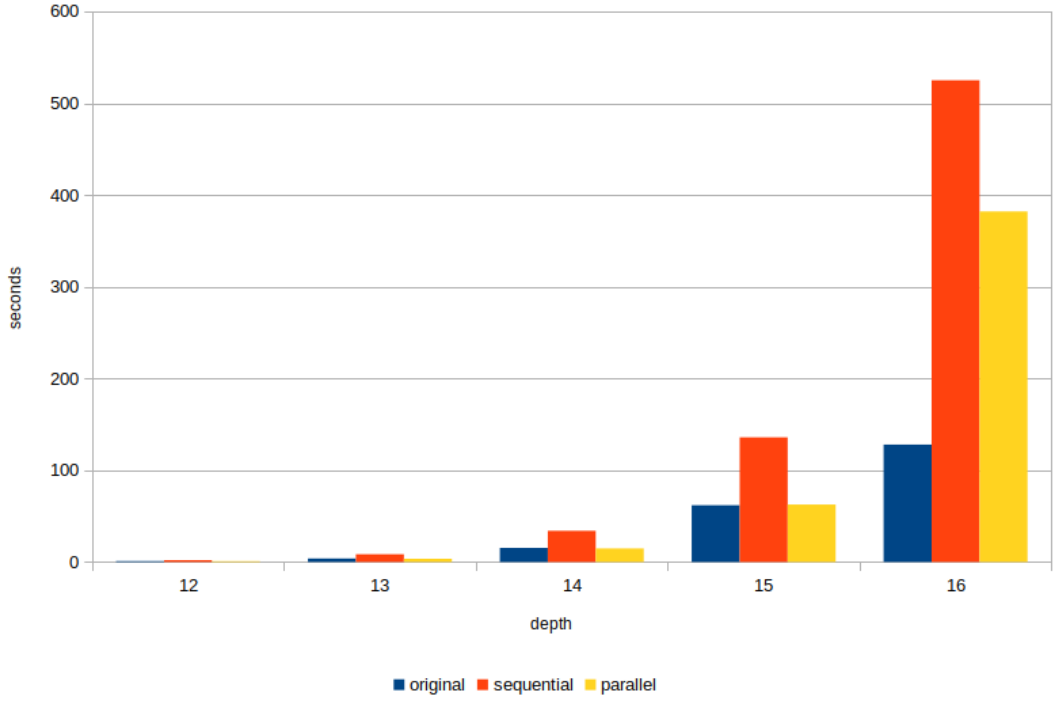

Figure 5.8: Library performance on the *perimeter* benchmark. Horizontal axis plots the depth of the perimeter tree (see Section 2.8.3). Vertical axis shows the time it takes to compute a single perimeter value with various tree depths (splitting granularity).

5.5 Limitations and Future work

5.5.1 Limitations

The limitations of our computational frameworks largely come as the implications of unconstrained allowance of side effects. Compared to pure functional patterns our

computational frameworks give a programmer more freedom by allowing to leave side effects, but this comes at the price of programmers having more responsibility. It is an interesting trade-off by itself, but a programmer still needs to understand computational patterns our frameworks support and a high-level parallelization they do to prevent possible race conditions. Aside conforming to a certain interface, our computational frameworks do not impose any particular restrictions on operator functions and provide a programmer with a freedom. Obviously, our computational frameworks are not universal and are limited to only those problems they are applicable to.

5.5.2 Future work

Chapter 3 presents an overview of related work on various automatic and semi-automatic software parallelization methods. Automatically parallelizing compilers are challenged by the limitations of static program analysis. As a workaround solution, various researchers have proposed techniques on automatic data structure and algorithmic skeleton recognition. The application of these techniques for the real world code could be exacerbated by the problem of algorithm and data structure inseparability.

The concept of computational frameworks and the prototype library could lay the foundation towards an alternative automatic parallelization technique. We believe that it is possible to automatically recognize computational frameworks in the suite of Olden benchmarks. Alike the work [76], the recognition technology could be based on the static analysis of the program's AST. Figure 5.9 shows the scheme. It takes an original legacy C source code, where we supposedly have a computation that fits into one or several of our frameworks. The task of the recognizer is to identify a computational pattern. Like in the case of the *health* Olden benchmark we have a recursive *grow()* and *compute()* procedures, which build and process the tree correspondingly. That recursive pattern can be matched to a fractal framework. The next step would be to strip the code corresponding and implementing the pattern (*backbone logic*) and leave the rest as the *business logic*. The business logic must be further classified into various computational framework template class methods (like *grow()*, *growth_stop_condition()*, *inject()*, etc.). Then, the class template must be instantiated with the business logic inserted into the right places. Once that is done, the parallelization is done, as the compiler will take care of everything else.

Technically, for benchmarks as simple as Olden the technique can work on the level

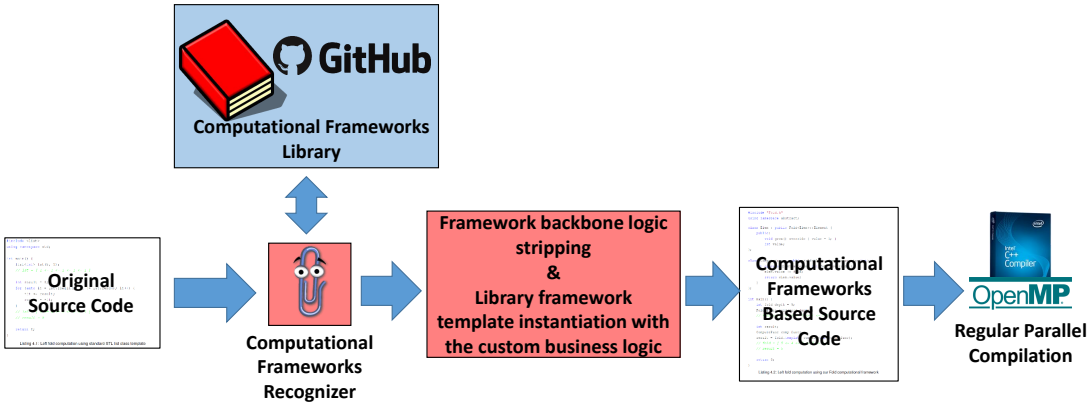

Figure 5.9: An alternative software parallelization scheme based on our library of computational frameworks.

of the compiler's front end: source code or an abstract syntax tree (AST). That sets the technique apart from the regular parallelization approaches based on dependence graphs and working on the level of the compiler's intermediate representation (IR). For more complex applications dynamic trace-based techniques could be used.

We envision this work as a possible future direction.

Chapter 6

Summary & Conclusions

Parallelism has become pervasive in the modern computing world. Parallel hardware is everywhere, but to exploit the available resources software has to be effectively mapped onto that hardware. The areas of *software parallelization* and *parallel software development* are extremely important. Despite decades of academic research and industrial investment into the area, the human expert still has a major role to play. Our state-of-the-art literature review shows that there are still no automatic solutions that could fully replace an expert programmer and at the same time guarantee program correctness and achieve performance results comparable to ”handmade” parallel software.

To achieve the best result a programmer has to work on multiple conceptual levels starting from problem decomposition and algorithm choice down to low-level loop transformations. In this thesis, we fully acknowledge the role of the human expert, but provide the latter with an *assistant solution*, which aims at alleviating the software parallelization and parallel software development tasks. The solution is as multifaceted as the problem itself, and also spans several conceptual levels. It consists of two components: a tool plus methodology and a library of parallel primitives. The tool aims to alleviate the task of sequential software parallelization by guiding a programmer through the process. The library can be used as a set of ready parallel solutions for specific problems.

The tool aims to assist human experts by guiding them directly towards the most interesting program loops from the perspective of software parallelization, thus alleviating the parallelizable code search process and delivering savings for this costly human resource. We have developed a novel machine learning based approach to predicting whether or not a loop is parallelizable. We combine this prediction with traditional

profiling information and develop a ranking function that prioritizes low-risk, high-gain loop candidates, which are finally presented to the user.

We have evaluated our parallelization assistant against the sequential C implementations of the SNU NPB suite. We show that our assistant recognizes parallelizable loops more aggressively than conservative parallelizing compilers, thus improving parallelism discovery. We also show that our parallelization assistant can increase programmer productivity. Our experiments confirm, that equipped with our assistant, a programmer is required to examine and parallelize substantially fewer loops to achieve performance levels comparable to those of the reference OpenMP implementations of the benchmarks.

But the most important, our work has demonstrated that there is scope for machine learning based tool support in parallelization despite its inherent lack of safety. By assisting human programmers rather than replacing them, machine learning techniques have the potential to deliver productivity gains beyond what is possible by relying on traditional parallelization approaches alone.

The second component of the assistant solution is the concept of *computational frameworks* along with a research prototype library implementing it. The problem of successful data structure choice stands particularly important and can vastly affect the parallelizability of programs. Moreover, we observe *the problem of algorithm and data structure inseparability*. These issues have led us to a novel concept of computational frameworks, which are higher-level entities that embody algorithms and data structures into an elegant and well-structured construct. Computational frameworks can be used as parallel software design and construction primitives alleviating the task and ultimately parallelizing a wide class of applications, which fit into their computational patterns.

We shaped the concept and designed the library using a subset of the Olden benchmarks. We expressed benchmark computations through our *Fractal*, *Fold* and *Reduce* frameworks and rewrote the benchmarks in a modern, well-structured, and crucially parallel way combining the elements of both object-oriented and functional programming. Moreover, the rejuvenated C++ benchmark versions demonstrate a good parallel performance compared to their serial legacy C counterparts. On the major benchmarks, we achieve 5-6x speedups. However, given the research prototype nature of the library, some further engineering effort is still needed.

In this thesis we demonstrate that when decades-old and well-known methods of software parallelization such as various automatic techniques run into their limits and

fail to tackle the challenges of the real-world code, and more exotic methods of machine learning based techniques run into their principal problems of inherent statistical errors and the lack of safety, it is possible to acknowledge the role of a human expert and resort to various assisting solutions. The latter demonstrates promising results and paves an attractive future research direction.

Bibliography

[1] LLVM Developer Group. *The LLVM Compiler Infrastructure*. 2002-2019. URL: https://llvm.org/.

[2] Aleksandr Maramzin. *Pervasive Parallelism Tool*. 2018-2019. URL: https://github.com/av-maramzin/icc-opt-report-compiler.

[3] Intel Corporation. *Intel C/C++ Compiler (ICC)*. 1985-2018. URL: https://software.intel.com/en-us/c-compilers.

[4] Gene M. Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). Atlantic City, New Jersey: Association for Computing Machinery, 1967, pp. 483–485. ISBN: 9781450378956. DOI: 10.1145/1465482.1465560. URL: https://doi.org/10.1145/1465482.1465560.

[5] T.J. McCabe. "A Complexity Measure". In: *IEEE Transactions on Software Engineering* SE-2.4 (1976), pp. 308–320. DOI: 10.1109/TSE.1976.233837.

[6] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. "The Program Dependence Graph and Its Use in Optimization". In: *ACM Trans. Program. Lang. Syst.* 9.3 (July 1987), pp. 319–349. ISSN: 0164-0925. DOI: 10.1145/24039.24041. URL: http://doi.acm.org/10.1145/24039.24041.

[7] Murray Cole. "Algorithmic Skeletons: Structured Management of Parallel Computation. Research Monographs in Parallel and Distributed Computing." In: MIT Press, Cambridge, 1991.

[8] Mary W. Hall, Ken Kennedy, and Kathryn S. McKinley. "Interprocedural Transformations for Parallel Code Generation". In: *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. Supercomputing '91. Albuquerque, New Mexico, USA: Association for Computing Machinery, 1991, pp. 424–434. ISBN:

0897914597. DOI: 10.1145/125826.126055. URL: https://doi.org/10.1145/125826.126055.

[9] M.E. Wolf and M.S. Lam. "A loop transformation theory and an algorithm to maximize parallelism". In: *IEEE Transactions on Parallel and Distributed Systems* 2.4 (1991), pp. 452–471. DOI: 10.1109/71.97902.

[10] Mary W. Hall et al. "FIAT: A Framework for Interprocedural Analysis and Transfomation". In: *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*. Berlin, Heidelberg: Springer-Verlag, 1993, pp. 522–545. ISBN: 3540576592.

[11] Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. "Array-Data Flow Analysis and Its Use in Array Privatization". In: *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '93. Charleston, South Carolina, USA: Association for Computing Machinery, 1993, pp. 2–15. ISBN: 0897915607. DOI: 10.1145/158511.158515. URL: https://doi.org/10.1145/158511.158515.

[12] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. "Compiler Transformations for High-performance Computing". In: *ACM Comput. Surv.* 26.4 (Dec. 1994), pp. 345–420. ISSN: 0360-0300. DOI: 10.1145/197405.197406. URL: http://doi.acm.org/10.1145/197405.197406.

[13] William Blume et al. "Automatic Detection of Parallelism: A Grand Challenge for High-Performance Computing". In: *IEEE Parallel Distrib. Technol.* 2.3 (Sept. 1994), pp. 37–47. ISSN: 1063-6552. DOI: 10.1109/M-PDT.1994.329796. URL: https://doi.org/10.1109/M-PDT.1994.329796.

[14] René Dekker and Frans Ververs. "Abstract Data Structure Recognition". In: *Proceedings of the 9th International Conference on Knowledge-Based Software Engineering*. KBSE'94. Monterey, CA, USA: IEEE Press, 1994, pp. 133–140. ISBN: 0-8186-6380-4. DOI: 10.1109/KBSE.1994.342669. URL: https://doi.org/10.1109/KBSE.1994.342669.

[15] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. "Context-sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers". In: *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*. PLDI '94. Orlando, Florida, USA: ACM, 1994,

pp. 242–256. ISBN: 0-89791-662-X. DOI: 10.1145/178243.178264. URL: http://doi.acm.org/10.1145/178243.178264.

[16] Robert Wilson et al. *The SUIF Compiler System: A Parallelizing and Optimizing Research Compiler*. Tech. rep. Stanford, CA, USA, 1994.

[17] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0201633612.

[18] Rakesh Ghiya and Laurie J. Hendren. "Is It a Tree, a DAG, or a Cyclic Graph? A Shape Analysis for Heap-directed Pointers in C". In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '96. St. Petersburg Beach, Florida, USA: ACM, 1996, pp. 1–15. ISBN: 0-89791-769-3. DOI: 10.1145/237721.237724. URL: http://doi.acm.org/10.1145/237721.237724.

[19] M.W. Hall et al. "Maximizing multiprocessor performance with the SUIF compiler". In: *Computer* 29.12 (1996), pp. 84–89. DOI: 10.1109/2.546613.

[20] Amy W. Lim and Monica S. Lam. "Maximizing Parallelism and Minimizing Synchronization with Affine Transforms". In: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '97. Paris, France: Association for Computing Machinery, 1997, pp. 201–214. ISBN: 0897918533. DOI: 10.1145/263699.263719. URL: https://doi.org/10.1145/263699.263719.

[21] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. ISBN: 1-55860-320-4.

[22] Leonardo Dagum and Ramesh Menon. "OpenMP: An Industry-Standard API for Shared-Memory Programming". In: *IEEE Comput. Sci. Eng.* 5.1 (Jan. 1998), pp. 46–55. ISSN: 1070-9924. DOI: 10.1109/99.660313. URL: https://doi.org/10.1109/99.660313.

[23] Lance Hammond, Mark Willey, and Kunle Olukotun. "Data Speculation Support for a Chip Multiprocessor". In: *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS VIII. San Jose, California, USA: Association for Com-

puting Machinery, 1998, pp. 58–69. ISBN: 1581131070. DOI: 10.1145/291069.291020. URL: https://doi.org/10.1145/291069.291020.

[24] J. Steffan and T Mowry. “The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization”. In: *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*. HPCA ’98. USA: IEEE Computer Society, 1998, p. 2. ISBN: 0818683236.

[25] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. “Parametric Shape Analysis via 3-valued Logic”. In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’99. San Antonio, Texas, USA: ACM, 1999, pp. 105–118. ISBN: 1-58113-095-3. DOI: 10.1145/292540.292552. URL: http://doi.acm.org/10.1145/292540.292552.

[26] Reinhard Wilhelm, Shmuel Sagiv, and Thomas W. Reps. “Shape Analysis”. In: *Proceedings of the 9th International Conference on Compiler Construction*. CC ’00. London, UK, UK: Springer-Verlag, 2000, pp. 1–17. ISBN: 3-540-67263-X. URL: http://dl.acm.org/citation.cfm?id=647476.760384.

[27] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002. ISBN: 1-55860-286-0.

[28] L. Almagor et al. “Finding Effective Compilation Sequences”. In: *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. LCTES ’04. Washington, DC, USA: ACM, 2004, pp. 231–239. ISBN: 1-58113-806-7. DOI: 10.1145/997163.997196. URL: http://doi.acm.org/10.1145/997163.997196.

[29] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. CGO ’04. Palo Alto, California: IEEE Computer Society, 2004, pp. 75–. ISBN: 0-7695-2102-9. URL: http://dl.acm.org/citation.cfm?id=977395.977673.

[30] R. Stahl et al. “High-level data-access analysis for characterisation of (sub)task-level parallelism on Java”. In: *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings.* 2004, pp. 31–40. DOI: 10.1109/HIPS.2004.1299188.

[31] Peng Zhao and José Nelson Amaral. "To Inline or Not to Inline? Enhanced Inlining Decisions". In: *Languages and Compilers for Parallel Computing*. Ed. by Lawrence Rauchwerger. Berlin, Heidelberg: Springer, 2004, pp. 405–419. ISBN: 978-3-540-24644-2. DOI: 10.1007/978-3-540-24644-2_26.

[32] J. Cavazos and M. F. P. O'Boyle. "Automatic Tuning of Inlining Heuristics". In: *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, Nov. 2005, pp. 14–14. DOI: 10.1109/SC.2005.14.

[33] Keith D. Cooper et al. "ACME: Adaptive Compilation Made Efficient". In: *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. LCTES '05. Chicago, Illinois, USA: ACM, 2005, pp. 69–77. ISBN: 1-59593-018-3. DOI: 10.1145/1065910.1065921. URL: http://doi.acm.org/10.1145/1065910.1065921.

[34] Alexandru Niculescu-Mizil and Rich Caruana. "Predicting Good Probabilities with Supervised Learning". In: *Proceedings of the 22nd International Conference on Machine Learning*. ICML '05. Bonn, Germany: ACM, 2005, pp. 625–632. ISBN: 1-59593-180-5. DOI: 10.1145/1102351.1102430. URL: http://doi.acm.org/10.1145/1102351.1102430.

[35] G. Ottoni et al. "Automatic thread extraction with decoupled software pipelining". In: *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*. 2005, 12 pp.–118. DOI: 10.1109/MICRO.2005.13.

[36] M. Stephenson and S. Amarasinghe. "Predicting unroll factors using supervised classification". In: *International Symposium on Code Generation and Optimization*. Washington, DC, USA: IEEE Computer Society, Mar. 2005, pp. 123–134. DOI: 10.1109/CGO.2005.29.

[37] Saisanthosh Balakrishnan and Gurindar S. Sohi. "Program Demultiplexing: Data-Flow Based Speculative Parallelization of Methods in Sequential Programs". In: *SIGARCH Comput. Archit. News* 34.2 (May 2006), pp. 302–313. ISSN: 0163-5964. DOI: 10.1145/1150019.1136512. URL: https://doi.org/10.1145/1150019.1136512.

[38] Wei Liu et al. "POSH: A TLS Compiler That Exploits Program Structure". In: *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '06. New York, New York, USA:

Association for Computing Machinery, 2006, pp. 158–167. ISBN: 1595931899. DOI: 10.1145/1122971.1122997. URL: https://doi.org/10.1145/1122971.1122997.

[39] "Exploiting Postdominance for Speculative Parallelization". In: *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. 2007, pp. 295–305. DOI: 10.1109/HPCA.2007.346207.

[40] Ram Rangan et al. "Performance Scalability of Decoupled Software Pipelining". In: *ACM Trans. Archit. Code Optim.* 5.2 (Sept. 2008). ISSN: 1544-3566. DOI: 10.1145/1400112.1400113. URL: https://doi.org/10.1145/1400112.1400113.

[41] Changhee Jung and Nathan Clark. "DDT: design and evaluation of a dynamic program analysis for optimizing data structure usage". In: *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. New York, New York: ACM, 2009, pp. 56–66. ISBN: 978-1-60558-798-1. DOI: http://doi.acm.org/10.1145/1669112.1669122.

[42] H. Leather, E. Bonilla, and M. O'Boyle. "Automatic Feature Generation for Machine Learning Based Optimizing Compilation". In: *2009 International Symposium on Code Generation and Optimization*. Washington, DC, USA: IEEE Computer Society, Mar. 2009, pp. 81–91. DOI: 10.1109/CGO.2009.21.

[43] Yueqiang Shang. "A distributed memory parallel Gauss–Seidel algorithm for linear algebraic systems". In: *Computers Mathematics with Applications* 57.8 (2009), pp. 1369–1376. ISSN: 0898-1221. DOI: https://doi.org/10.1016/j.camwa.2009.01.034. URL: https://www.sciencedirect.com/science/article/pii/S089812210900042X.

[44] Naeem Abbas et al. "Accelerating HMMER on FPGA using parallel prefixes and reductions". In: Aug. 2010, pp. 37–44. DOI: 10.1109/FPT.2010.5681755.

[45] Narayan Ganesan et al. "Accelerating HMMER on GPUs by Implementing Hybrid Data and Task Parallelism". In: *Proceedings of the First ACM International Conference on Bioinformatics and Computational Biology*. BCB '10. Niagara Falls, New York: ACM, 2010, pp. 418–421. ISBN: 978-1-4503-0438-2. DOI: 10.1145/1854776.1854844. URL: http://doi.acm.org/10.1145/1854776.1854844.

[46] Christopher Mims. “MIT Technology Review. Why CPUs Aren’t Getting Any Faster. Making computers faster means relying on the central processing unit (CPU) less than ever before.” In: (2010). URL: https://www.technologyreview.com/2010/10/12/199966/why-cpus-arent-getting-any-faster/.

[47] Sean Rul, Hans Vandierendonck, and Koen De Bosschere. “A Profile-Based Tool for Finding Pipeline Parallelism in Sequential Programs”. In: *Parallel Comput.* 36.9 (Sept. 2010), pp. 531–551. ISSN: 0167-8191. DOI: 10.1016/j.parco.2010.05.006. URL: https://doi.org/10.1016/j.parco.2010.05.006.

[48] Marcin Wozniak, Tomasz Olas, and Roman Wyrzykowski. “Parallel Implementation of Conjugate Gradient Method on Graphics Processors”. In: *Parallel Processing and Applied Mathematics*. Ed. by Roman Wyrzykowski et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 125–135. ISBN: 978-3-642-14390-8.

[49] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[50] Sangmin Seo, Gangwon Jo, and Jaejin Lee. “Performance Characterization of the NAS Parallel Benchmarks in OpenCL”. In: *Proceedings of the 2011 IEEE International Symposium on Workload Characterization*. IISWC ’11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 137–148. ISBN: 978-1-4577-2063-5. DOI: 10.1109/IISWC.2011.6114174. URL: http://dx.doi.org/10.1109/IISWC.2011.6114174.

[51] NASA Advanced Supercomputing (NAS) Division. *NAS Parallel Benchmarks*. Aug. 2012. URL: https://www.nas.nasa.gov/publications/npb.html.

[52] Rudi Helfenstein and Jonas Koko. “Parallel preconditioned conjugate gradient algorithm on GPU”. In: *Journal of Computational and Applied Mathematics* 236.15 (2012). Proceedings of the Fifteenth International Congress on Computational and Applied Mathematics (ICCAM-2010), Leuven, Belgium, 5-9 July, 2010, pp. 3584–3590. ISSN: 0377-0427. DOI: https://doi.org/10.1016/j.cam.2011.04.025. URL: https://www.sciencedirect.com/science/article/pii/S0377042711002196.

[53] Per Larsen et al. “Parallelizing More Loops with Compiler Guided Refactoring”. In: *Proceedings of the 2012 41st International Conference on Parallel Processing*. ICPP ’12. Washington, DC, USA: IEEE Computer Society, 2012,

pp. 410–419. ISBN: 978-0-7695-4796-1. DOI: 10.1109/ICPP.2012.48. URL: http://dx.doi.org/10.1109/ICPP.2012.48.

[54] Michael McCool, James Reinders, and Arch Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012. ISBN: 9780123914439.

[55] Samuel Midkiff. *Automatic Parallelization: An Overview of Fundamental Compiler Techniques*. 2012.

[56] Herb Sutter. "Welcome to the Jungle! Or, A Heterogeneous Supercomputer in Every Pocket". In: (2012). URL: https://herbsutter.com/welcome-to-the-jungle/.

[57] Seoul National University. *SNU NAS Parallel Benchmarks*. Aug. 2012. URL: http://aces.snu.ac.kr/software/snu-npb/.

[58] Daniel Fried et al. "Predicting Parallelization of Sequential Programs Using Supervised Learning". In: *Proc. of the 12th IEEE International Conference on Machine Learning and Applications (ICMLA), Miami, FL, USA*. Miami, FL, USA: IEEE Computer Society, Dec. 2013, pp. 72–77. DOI: 10.1109/ICMLA.2013.108. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6786084.

[59] Gareth James et al. *An introduction to statistical learning : with applications in R*. Heidelberg, Germany: Springer, 2013. ISBN: 978-1-4614-7138-7. DOI: 10.1007/978-1-4614-7138-7.

[60] Girish Sharma, Abhishek Agarwala, and Baidurya Bhattacharya. "A fast parallel Gauss Jordan algorithm for matrix inversion using CUDA". In: *Computers Structures* 128 (2013), pp. 31–37. ISSN: 0045-7949. DOI: https://doi.org/10.1016/j.compstruc.2013.06.015. URL: https://www.sciencedirect.com/science/article/pii/S0045794913002095.

[61] Istvan Bozo et al. "Discovering Parallel Pattern Candidates in Erlang". In: *Proceedings of the Thirteenth ACM SIGPLAN Workshop on Erlang*. Erlang '14. Gothenburg, Sweden: Association for Computing Machinery, 2014, pp. 13–23. ISBN: 9781450330381. DOI: 10.1145/2633448.2633453. URL: https://doi.org/10.1145/2633448.2633453.

[62] "Fastflow: High-Level and Efficient Streaming on Multicore". In: Mar. 2014. ISBN: 9780470936900. DOI: 10.1002/9781119332015.ch13.

[63] Akihiro Hayashi et al. “Machine-Learning-based Performance Heuristics for Runtime CPU/GPU Selection”. In: *Proceedings of the Principles and Practices of Programming on The Java Platform*. PPPJ ’15. Melbourne, FL, USA: ACM, 2015, pp. 27–36. ISBN: 978-1-4503-3712-0. DOI: 10.1145/2807426.2807429. URL: http://doi.acm.org/10.1145/2807426.2807429.

[64] Istvan Haller, Asia Slowinska, and Herbert Bos. “Scalable Data Structure Detection and Classification for C/C++ Binaries”. In: *Empirical Softw. Engg.* 21.3 (June 2016), pp. 778–810. ISSN: 1382-3256. DOI: 10.1007/s10664-015-9363-y. URL: http://dx.doi.org/10.1007/s10664-015-9363-y.

[65] Kiminori Matsuzaki and Reina Miyazaki. “Parallel Tree Accumulations on MapReduce”. In: *Int. J. Parallel Program.* 44.3 (June 2016), pp. 466–485. ISSN: 0885-7458. DOI: 10.1007/s10766-015-0355-8. URL: https://doi.org/10.1007/s10766-015-0355-8.

[66] “RPL: A Domain-Specific Language for Designing and Implementing Parallel C++ Applications”. In: *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*. 2016, pp. 288–295. DOI: 10.1109/PDP.2016.122.

[67] Amir H. Ashouri et al. “MiCOMP: Mitigating the Compiler Phase-Ordering Problem Using Optimization Sub-Sequences and Machine Learning”. In: *ACM Trans. Archit. Code Optim.* 14.3 (Sept. 2017), 29:1–29:28. ISSN: 1544-3566. DOI: 10.1145/3124452. URL: http://doi.acm.org/10.1145/3124452.

[68] Philip Ginsbach and Michael F. P. O'Boyle. “Discovery and Exploitation of General Reductions: A Constraint Based Approach”. In: *Proceedings of the 2017 International Symposium on Code Generation and Optimization*. CGO ’17. Austin, USA: IEEE Press, 2017, pp. 269–280. ISBN: 978-1-5090-4931-8. URL: http://dl.acm.org/citation.cfm?id=3049832.3049862.

[69] Nicklas Bo Jensen and Sven Karlsson. “Improving Loop Dependence Analysis”. In: *ACM Trans. Archit. Code Optim.* 14.3 (Aug. 2017), 22:1–22:24. ISSN: 1544-3566. DOI: 10.1145/3095754. URL: http://doi.acm.org/10.1145/3095754.

[70] Thomas Rupprecht et al. “DSIbin: Identifying Dynamic Data Structures in C/C++ Binaries”. In: *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*. ASE 2017. Urbana-Champaign, IL, USA:

IEEE Press, 2017, pp. 331–341. ISBN: 978-1-5386-2684-9. URL: http://dl.acm.org/citation.cfm?id=3155562.3155607.

[71] Chao Chen et al. “A distributed-memory hierarchical solver for general sparse linear systems”. In: *Parallel Computing* 74 (2018). Parallel Matrix Algorithms and Applications (PMAA’16), pp. 49–64. ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2017.12.004. URL: https://www.sciencedirect.com/science/article/pii/S0167819117302077.

[72] Mehmet Deveci, Christian Trott, and Sivasankaran Rajamanickam. “Multithreaded sparse matrix-matrix multiplication for many-core and GPU architectures”. In: *Parallel Computing* 78 (2018), pp. 33–46. ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2018.06.009. URL: https://www.sciencedirect.com/science/article/pii/S0167819118301923.

[73] Philip Ginsbach, Lewis Crawford, and Michael F. P. O'Boyle. “CAnDL: A Domain Specific Language for Compiler Analysis”. In: *Proceedings of the 27th International Conference on Compiler Construction*. CC 2018. Vienna, Austria: ACM, 2018, pp. 151–162. ISBN: 978-1-4503-5644-2. DOI: 10.1145/3178372.3179515. URL: http://doi.acm.org/10.1145/3178372.3179515.

[74] Philip Ginsbach et al. “Automatic Matching of Legacy Code to Heterogeneous APIs: An Idiomatic Approach”. In: *SIGPLAN Not.* 53.2 (Mar. 2018), pp. 139–153. ISSN: 0362-1340. DOI: 10.1145/3296957.3173182. URL: http://doi.acm.org/10.1145/3296957.3173182.

[75] Stanislav Manilov, Christos Vasiladiotis, and Björn Franke. “Generalized Profile-guided Iterator Recognition”. In: *Proceedings of the 27th International Conference on Compiler Construction*. CC 2018. Vienna, Austria: ACM, 2018, pp. 185–195. ISBN: 978-1-4503-5644-2. DOI: 10.1145/3178372.3179511. URL: http://doi.acm.org/10.1145/3178372.3179511.

[76] David del Rio Astorga et al. “Finding Parallel Patterns through Static Analysis in C++ Applications”. In: *Int. J. High Perform. Comput. Appl.* 32.6 (Nov. 2018), pp. 779–788. ISSN: 1094-3420. DOI: 10.1177/1094342017695639. URL: https://doi.org/10.1177/1094342017695639.

[77] Zheng Wang and Michael O’Boyle. “Machine Learning in Compiler Optimization”. English. In: *Proceedings of the IEEE* 106.11 (May 2018), pp. 1879–1901. ISSN: 0018-9219. DOI: 10.1109/JPROC.2018.2817118.

[78] Aleksandr Maramzin. *Machine Learning Based Parallelization Assistant*. The University of Edinburgh, 2019. URL: https://github.com/av-maramzin/PParMetrics.

[79] Aleksandr Maramzin et al. ""It Looks like You'Re Writing a Parallel Loop": A Machine Learning Based Parallelization Assistant". In: *Proceedings of the 6th ACM SIGPLAN International Workshop on AI-Inspired and Empirical Methods for Software Engineering on Parallel Computing Systems*. AI-SEPS 2019. Athens, Greece: Association for Computing Machinery, 2019, pp. 1–10. ISBN: 9781450369831. DOI: 10.1145/3358500.3361567. URL: https://doi.org/10.1145/3358500.3361567.

[80] Prema Soundararajan et al. "A study on popular auto-parallelization frameworks". In: *Concurrency and Computation: Practice and Experience* 31 (Feb. 2019), e5168. DOI: 10.1002/cpe.5168.

[81] "A Hybrid Approach to Parallel Pattern Discovery in C++". In: *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. 2020, pp. 187–191. DOI: 10.1109/PDP50117.2020.00035.

[82] Aleksandr Maramzin. *Computational Frameworks*. The University of Edinburgh, 2020. URL: https://github.com/av-maramzin/Abstract-DT.

[83] "Algorithmic Skeletons and Parallel Design Patterns in Mainstream Parallel Programming". In: International Journal of Parallel Programming, 2021. DOI: 10.1007/s10766-020-00684-w. URL: https://doi.org/10.1007/s10766-020-00684-w.

[84] ISO IEEE. *ISO/IEC/IEEE 9945:2009 Information technology — Portable Operating System Interface (POSIX®) Base Specifications, Issue 7*. 2021. URL: https://www.iso.org/standard/50516.html.

[85] Roberto Castaneda Lozano, Murray Cole, and Bjorn Franke. "Modernizing Parallel Code with Pattern Analysis". In: *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 418–430. ISBN: 9781450382946. DOI: 10.1145/3437801.3441603. URL: https://doi.org/10.1145/3437801.3441603.

[86] "Restoration of Legacy Parallelism: Transforming Pthreads into Farm and Pipeline Patterns". In: *International Journal of Parallel Programming* (2021). DOI: 10.1007/s10766-021-00716-z. URL: https://doi.org/10.1007/s10766-021-00716-z.

[87] OpenMP Architecture Review Board. *The OpenMP API specification for parallel programming*. URL: https://www.openmp.org/.

[88] MPI Forum. *Message Passing Interface*. URL: https://www.mpi-forum.org/.

[89] Apple Inc. *OpenCL (Open Computing Language)*. URL: https://opencl.org/.

[90] Intel. *Threading Building Blocks*. URL: http://software.intel.com/en-us/intel-tbb.

[91] Lawrence Livermore National Laboratory. *Parallel Computing Tutorials*. URL: https://computing.llnl.gov/.

[92] Microsoft. *Parallel Patterns Library (PPL)*. URL: https://docs.microsoft.com/en-us/cpp/parallel/concrt/parallel-patterns-library-ppl?view=msvc-160.

[93] Nvidia. *CUDA (Compute Unified Device Architecture)*. URL: https://developer.nvidia.com/cuda-zone.

[94] "perf: Linux profiling with performance counters." In: (). URL: https://perf.wiki.kernel.org/index.php/Main_Page.

[95] "Polaris: Automatic Parallelization of Conventional Fortran Programs." In: (). URL: http://polaris.cs.uiuc.edu/polaris/polaris-old.html.

[96] GNU Project. *GCC, the GNU Compiler Collection*. URL: https://gcc.gnu.org/.

[97] Computer Systems Laboratory Stanford University. *SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers*. URL: https://suif.stanford.edu/suif/suif1/suif-overview/suif.html.