SUPPORTING VISUAL DIAGNOSIS OF PERFORMANCE PROBLEMS IN MULTI-CORE AND PARALLEL SOFTWARE

Roman Atachiants

Thesis submitted for the Degree of Doctor of Philosophy

School of Computer Science & Statistics

Trinity College

University of Dublin

29 September 2015

DECLARATION

I declare that this thesis has not been submitted as an exercise for a degree at this or any other university and it is entirely my own work. Wherever there is published or unpublished work included, it is duly acknowledged in the text.

I agree to deposit this thesis in the University's open access institutional repository or allow the library to do so on my behalf, subject to Irish Copyright Legislation and Trinity College Library conditions of use and acknowledgement.

ABSTRACT

The shift towards multicore processing has led to a much wider population of developers being faced with the challenge of exploiting parallel cores to improve software performance. Debugging and optimising parallel programs is a complex and demanding task. Tools which support development of parallel programs should provide salient information to allow programmers of multicore systems to diagnose and distinguish performance problems. Appropriate design of such tools requires a systematic analysis of the problems which might be identified, and the information used to diagnose them.

In this dissertation we present a general framework aimed to support designers of parallel performance analysis tools. The framework consists of several major components including: general advice for tool developers, a parallel performance problem taxonomy, an observational model for "data-to-problem" mapping, a deeper analysis of a data locality problem identification and a visualisation tool which we have used to evaluate the effectiveness of the approach.

First, with the aim of identifying issues, emerging practices and design opportunities for support, we present in this dissertation a qualitative study in which we interviewed a range of software developers, in both industry and academia. We then perform a systematic analysis of the data and identify several cross-cutting themes. These analysis themes include the practical relevance of the probe effect, the significance of orchestration models in development and the mismatch between currently available tools and developers' needs. We also identify an important characteristic of parallel programming, where the process of optimisation goes hand in hand with the process of debugging, as opposed to clearer distinctions which may be made in traditional programming. We conclude with reflection on how the study can inform the design of software tools to support developers in the endeavour of parallel program-

ming.

Next, building on the literature, we put forward a potential taxonomy of parallel performance problems, and an observational model which links measurable performance data to these problems. We present a validation of this model carried out with parallel programming experts, identifying areas of agreement and disagreement. This is accompanied with a survey of the prevalence of these problems in software development. From this we can identify contentious areas worthy of further exploration, as well as those with high prevalence and strong agreement, which are natural candidates for initial moves towards better tool support.

Finally, in order to explore the design space and how the framework can be used in the design of visualisations to support performance optimisation, the specific case of data locality is examined in more detail, and a prototype visualisation to support the diagnosis of data locality problems is introduced. Furthermore, an empirical evaluation of the visualisation was performed and the results are discussed as we reflect on the implications for the support of multicore performance analysis.

DEDICATION

Dedicated to the memory of my teacher, advisor, colleague and dear friend, Bernard Tollet (1973-2015) who inspired, supported and encouraged me to pursue the Ph.D

ACKNOWLEDGEMENTS

Immeasurable appreciation and deepest gratitude for the help and support are extended to the following persons, who in one way or another have contributed in making this study possible.

First and foremost I would like express my sincere gratitude to my research advisor and HCI specialist, Dr. Gavin Doherty for the continuous support throughout my Ph.D study as well as his patience, motivation, and enthusiasm. His continuous support and guidance helped me not only complete this dissertation but also deepen my knowledge of Human Computer Interaction. I could not have asked for a better research advisor.

Besides my research advisor, I would like to thank Dr. David Gregg whose domain expertise in Computer Architecture has proven to be an invaluable asset and made this study possible. He has always been there to listen and give advice.

I would like to thank LERO, IBM Research and Science Foundation Ireland for providing me with an opportunity to pursue my Ph.D. I extend thanks to my fellow lab mates, especially to my dear friend Oscar Cassetti and Drs. Erwan Moreau, Stephan Schlögl, Bérenger Arnaud, Kim Jarvis, Liliana Mamani Sanchez for their help and encouragement.

This study would not be possible without over one hundred participants who took part in multiple experiments. Their time and insights were indispensable for completing the study.

I extend my gratitude to Dr. Marco La Civita, Prof. Mikhail Kosov and Prof. Eduard Hoenkamp for their letters of recommendation which allowed me to enrol in the Ph.D in the first place, and who also inspired me and encouraged me to continue my studies.

I would also like to thank my girlfriend, Victoria, for taking care of me, for her

reassurance, her proof reading skills and excellent cooking ability. Her support helped me to stay sane and focused.

In addition, I would like to thank my family: Tatiana, Ruslan and my young brother Allan for their patience and confidence in my abilities.

RELATED PUBLICATIONS

- **Roman Atachiants**, David Gregg, Kim Jarvis, and Gavin Doherty. 2014. Design considerations for parallel performance tools. *In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14).* ACM, New York, NY, USA, 2501-2510. DOI:10.1145/2556288.2557350. **Honorable Mention Award.**
- **Roman Atachiants**, David Gregg and Gavin Doherty. 2015. An Observational Model for Identifying Parallel Performance Problems. *IEEE Transactions on Software Engineering (TSE).* **Under Revision.**
- **Roman Atachiants**, David Gregg and Gavin Doherty. Visualising Data Locality Performance for Parallel Programming. 2016. **Submitted for Publication.**

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1 INTRODUCTION

1.1 MICROPROCESSOR EVOLUTION

As computers become more prevalent in modern society, the tasks that computers may perform have also become increasingly complicated. In order to cope with this complexity, programmers develop ever more computationally demanding algorithms and applications. Likewise the volume of data processed by computers has also increased enormously. These factors together have led to ever expanding demands on microprocessor performance. In order to cope with this growing demand for computational power, hardware manufacturers continually increased the clock frequency of their central processing units (CPUs). However, this approach meant that the power consumption of each CPU also trended upwards. This strategy of increasing frequency eventually became less viable, as the power required to improve performance introduced a range of difficulties, such as excessive heat generation and leakage current.

An alternative strategy for increasing the number of instructions per second that a CPU can process is to put multiple processors (cores) on the same chip. Many modern personal computers now have two or more cores that enable multiple tasks (threads) to be executed simultaneously. This concerns not only servers and supercomputers, but also concerns any possible variety of computers: from smart-phones and game consoles to laptops and tablets.

Figure 1.1 depicts the historical evolution of the commodity CPUs. While the clock speed per core has stabilised around 3 GHz, the number of cores increases steadily, thus the total number of transistors on each CPU is effectively following *Moore's law*, the observation by Gordon Moore, co-founder of Intel, who predicted back in 1965 that the number of transistors per square inch on integrated circuits would double

every year for the foreseeable future.

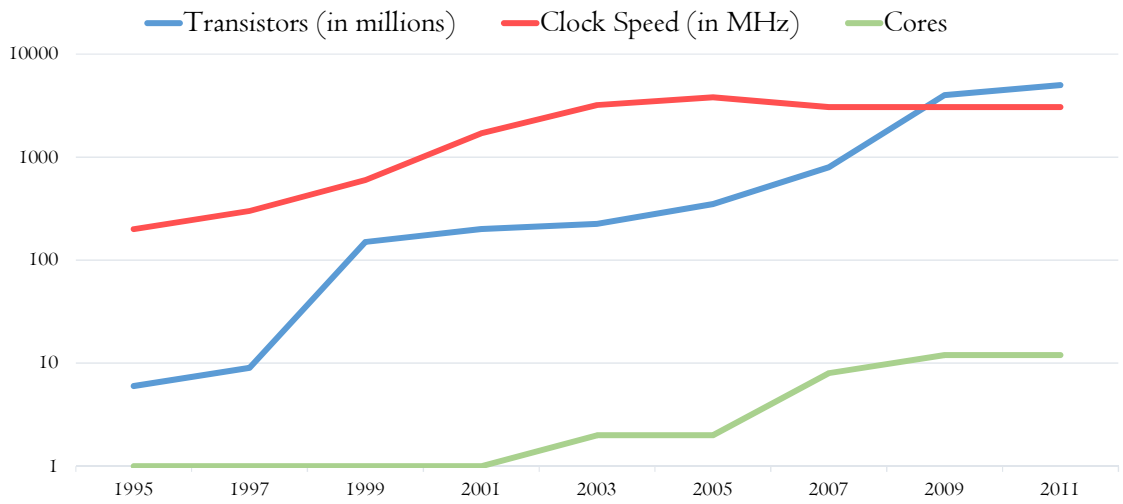

Figure 1.1 – Historical evolution of the clock speed, amount of transistors and number of cores per CPU.

On modern servers it is already very common to have more than 10 cores per CPU. For example, Intel Xeon E5 has 18 cores per socket with hyper-threading technology that effectively doubles the amount of cores for the operating system and applications. The table 1.1 shows the evolution of Intel processors for the past three decades. It is interesting to note that the cache and memory evolution seems to evolve slower than the total number of the transistors.

While the sheer number of transistors keeps increasing and doubling roughly every three years, as Moore's law predicts, the speed of the main memory while growing, is growing significantly slower than the computational capability of the CPUs. This rising disparity between the speed of the main memory and the CPU is commonly known as the **memory wall** [172]. While back in the 1980's a simple access to memory would be equivalent to a single CPU cycle, today the same access would cost between 50-300 cycles depending on the architecture, and this number is rising. This problem has been acknowledged and has been addressed over the past 20 years through various strategies, including the design of more data-aware processors and optimisation of the compilers that we use [29].

In the near future, computers are expected to have even more cores - the trend towards "many-core" computing. A many-core processor is a multi-core processor in which the number of cores is large enough that traditional multiprocessor techniques

Processor	Date	Clock	Threads	Level 1	Level 2	Level 3
8086	1978	8 MHz	1			
Intel 286	1982	12.5 MHz	1			
Intel 386 DX	1985	20 MHz	1			
Intel 486 DX	1989	25 MHz	1	8 KB		
Pentium	1993	60 MHz	1	16 KB		
Pentium Pro	1995	200 MHz	1	16 KB	256/512 KB	
Pentium 2	1997	266 MHz	1	32 KB	256/512 KB	
Pentium 3	1999	500 MHz	1	32 KB	512 KB	
Xeon	2001	1.7 GHz	1	8 KB	512 KB	
Pentium M	2003	1.6 GHz	1	64 KB	1 MB	
Xeon X5355	2006	2.67 GHz	4	4 x 32 KB	2 x 4 MB	
Xeon X7460	2008	2.67 GHz	6	6 x 32 KB	3 x 3 MB	16 MB
Xeon X7560	2010	2.26 GHz	16	8 x 64 KB	8 x 256 KB	24 MB
Xeon E7-8870	2011	2.4 GHz	20	10 x 64 KB	10 x 256 KB	30 MB
Xeon E7-8870 v2	2013	2.3 GHZ	30	15 x 32 KB	15 x 256 KB	30 MB
Xeon E7-8870 v3	2015	2.1 GHz	36	18 x 32 KB	18 x 256 KB	45 MB

Table 1.1 – The evolution of Intel processors in the past 35 years.

are no longer efficient. While with a small number of cores, performance gains can be achieved simply by running different programs simultaneously; with many cores, performance gains will only be achieved through the use of parallel programming.

One of the technologies that could lead us to the thousands of cores per CPU are known as **three dimensional integrated circuits** (3D IC) where the cores are no longer simply located next to each other on a horizontal pane, but stacked vertically, which significantly reduces the electric power requirements along with heat and increased efficiency.

1.2 PARALLEL PROGRAMMING

In order to take advantage of the multi-core and many-core hardware of today and tomorrow, programmers are faced with a need to parallelise their code and distribute work across multiple processors. This process of parallelisation is complex and requires application programmers to think about many possible outcomes and situations that may occur.

Traditionally, software has been written for **serial** computation, where a problem is broken down into a discrete series of instructions and those instructions are executed

sequentially, one after another by a single processor. This leads to a single instruction being executed at a time and the programmer can safely assume that the previous instruction has completed before the new instruction begins execution. This process is illustrated in the Figure 1.2.

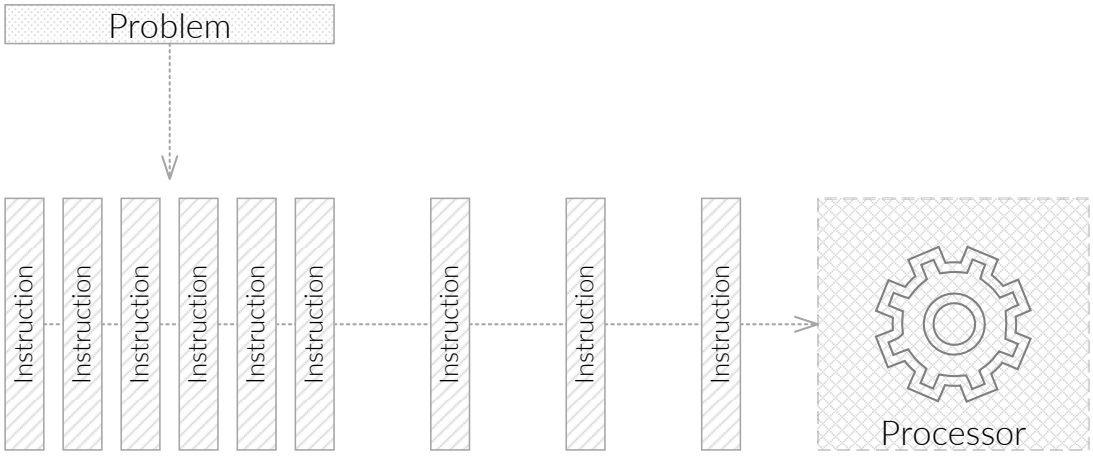

Figure 1.2 – Abstract representation of serial computing, where a problem is broken up into smaller pieces that are executed serially.

On the other hand, **parallel computing** is the simultaneous use of multiple computational units (processors) to solve a computational problem. In order to be effectively parallelised, the computational problem should be able to:

- Be broken apart into discrete sub-tasks, pieces of work that can be computed simultaneously.
- Execute multiple program instructions at any moment in time, as depicted in the Figure 1.3.
- Be solved in less time with multiple compute resources than with a single compute resource.

The compute resources can vary, but usually are a single computer with multiple processors (CPUs), a single processor with multiple cores or an arbitrary number of such computers connected by some kind of network to be able to communicate together for coordination purposes.

As the programmers seek to develop parallel applications, they encounter several major challenges. Those challenges are present on both: multi-core CPUs and GPUs,

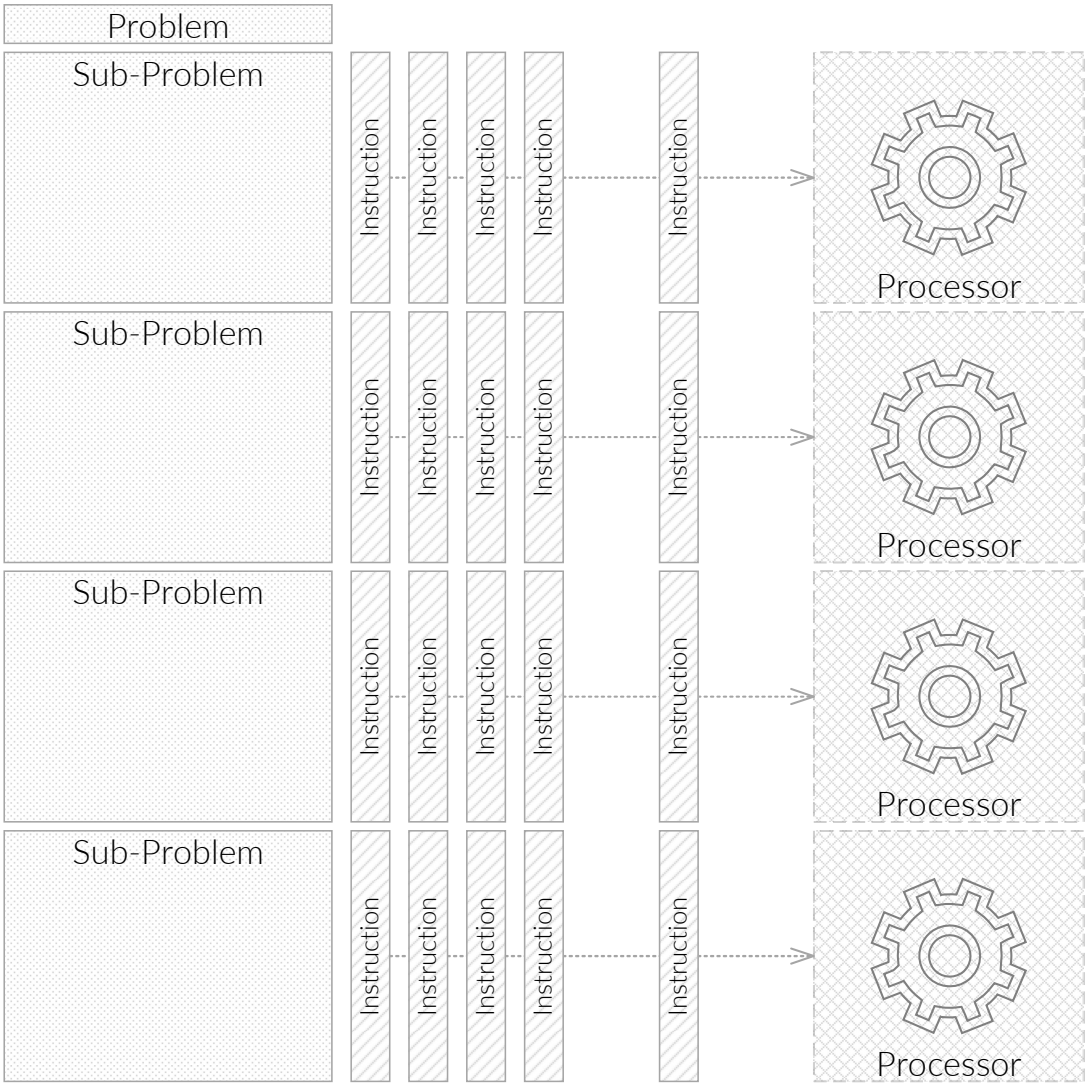

Figure 1.3 – Abstract representation of parallel computing, where a problem is broken up in smaller pieces that are executed in parallel by several processing units.

and the programmers themselves must address them in their applications. Such challenges are the reasons why compiler-based solutions from vendors will have limited success in creating a scalable parallel code base for many applications, as argued by Wen-mei Hwu, Parallel Computing Institute Chief Scientist of the University of Illinois. He contends that one of the main challenges in parallel programming is the high engineering effort required in the implementation of efficient parallel algorithms with high levels of parallelism and good data locality.

A sparse matrix-vector multiplication kernel is a good example of such a problem where there is ample parallelism in the algorithm, but little data to reuse due to the

vector set being simply too large to be stored in the memory, resulting in computation limited by the memory accesses.

This example is widely used in research on parallel performance tuning, while having a very simple implementation consisting of just a few lines of code. However, the reality is that even ordinary programmers have to deal with parallel systems with orders of magnitude of larger complexity than this matrix multiplication example. Numerous levels of indirection in the real systems while hiding the complexity might also obfuscate in-memory data layouts that are crucial for designing efficient and highly parallel algorithms.

Currently, the parallel programmer needs to determine layout arrangements of data, allocate memory and temporary storage, arrange pointers, perform index calculation, and orchestrate data movement in order to make use of the on-chip memory resources to support data re-use.

The programmer also has to decompose work into tasks, organise threads to perform the tasks, perform thread index calculations to access data in different levels of the memory hierarchy, determine data sharing patterns, and check data bounds. Many parameters of these arrangements need to be determined for each hardware platform. All such tasks are complicated and error prone, moreover the implementation also depends greatly on the target hardware platform and unfortunately there is not much compiler technology available today to address these challenges.

In this thesis, we aim to provide insights that will lower the engineering effort required for understanding the vast complexity of information involved in the process of parallel programming and identifying performance problems. We hope to provide insights to allow ordinary developers to effectively identify potential problems and support the processes of parallelisation and further performance tuning.

1.3 ADDRESSING THE CHALLENGES

A programmer seeking to parallelise a program has to overcome the challenges of synchronisation, non-determinism and orchestration that a programmer writing an equivalent sequential program would not face. Additionally, the very process of parallelising may introduce bugs, deadlocks and race conditions into the program. On

top of that, when looking at a program it is not necessarily obvious what its parallel performance will be.

For over two decades, a great deal of research effort has been directed at tools for improving the performance of parallel applications and over 200 now defunct, parallel-programming languages saw the light in the 1990s [112]. However, twenty years later, concurrency bugs are still extremely common and theoretical performance is often very difficult to achieve [145]. According to the 2011 UBM TechWeb Survey of 275 software engineers or managers of development teams, 32% of participants spend 6+ hours in a month finding data races or deadlock conditions and 69% spend 6+ hours in a month tuning the performance of their applications [1].

In this thesis, we aim to address the challenges in parallel programming by investigating how tools for parallel programmers can be designed in order to support programmers more effectively. In particular, we aim to ease the engineering effort required for the implementation of efficient parallel algorithms by providing visualisation tools to support both expert and ordinary programmers in the task of designing and optimising parallel programs. In order to even consider building effective tools to support and ease the engineering effort, we must first understand the needs of programmers in the field, how they currently cope with the problems and the tools they use. Hence, we present a field study of the challenges faced by parallel programmers.

In addition to the fieldwork, this thesis also presents a taxonomy of parallel performance problems constructed during our investigation of the problem space, as to our surprise, no extensive taxonomy existed, while there are a multitude of performance tools. This raises questions on the validity of existing performance tools and how effectively they can support the performance problem diagnosis without a (semi-)comprehensive analysis of the problem space to base the design upon.

One of the problems we will explore in the following chapters of this thesis is the so-called 'data locality' problem. This problem, induced by the 'memory wall', is intimately intertwined with parallelism [84] and is one of the key issues in parallel programming with modern and widespread architectures [119]. When developers build parallel software, performance is usually one of the key goals, yet data locality is often just as important as parallelism for performance. For example, accessing

[1] Parallel programming landscape: `http://www.danysoft.com/free/Intelparallelprog.pdf`

data in main memory can take hundreds of times longer than accessing the same data from the first level cache [70]. Thus, the programmer needs to be able to identify data locality problems when they arise in parallel programs. In addition, parallel threads executing on different cores often share the same data in one or more levels of cache which can improve locality. Equally, the threads may end up competing to keep their own data within the cache. The result can be complex interactions which cause subtle locality problems that may be difficult for the developer to identify.

Taking the parallel sparse matrix-vector multiplication implementations as a baseline for our data locality investigation, we examine the necessary information for the construction of effective displays that allow diagnosis of data locality problems. This comes with its own set of challenges, as we will need to look into low-level hardware performance counters and find a way of extracting useful metrics which we can later display. We present a model that brings together both analytical and technical aspects and allows us to identify interesting metrics, among thousands we can potentially collect, for each of the performance problems in our taxonomy.

Finally, we present an interactive visualisation tool for data locality identification, having performed a significant amount of fieldwork and analytical work in order to simply answer the question of *what to show on the screen?*, along with an evaluation of the tool using a set of parallel programs exhibiting performance issues with regard to data locality.

In this thesis we will be building towards a general framework for tool designers and consider many different aspects related to this - from understanding field practices to effective visualisation metaphors and ways of collecting and analysing relevant hardware performance data.

While the research aspirations are challenging, it is incumbent on us to distil the process into three major research questions. In this thesis, each chapter goes into more detail on each research question, but we use them to fundamentally guide the process of the research. The questions can be summarised in three words: **understand**, **model** and **visualise**.

- **RQ 1: How do people conceptualise parallel programs?**
- **RQ 2: Can we perform a comprehensive and systematic analysis of the information involved in understanding and improving the performance of parallel**

programs?

- **RQ 3: To what extent can a visualisation effectively support programmers in the task of optimising parallel programs?**

In the Chapter 3 we give further details on each research question and present an overview of how we have addressed each question. Answering to these three research questions allowed us to create a general framework that can be used to design various support tools for ordinary developers and aid them in the process of parallel performance problem diagnosis. Understanding how programmers conceptualise parallel programming, knowing the problems that can occur and the information important to diagnose those problems were crucial questions that required an answer prior to the creation of any software artefact to support programmers.

CHAPTER 2 RELATED WORK

The work presented in this thesis is situated at the intersection of **program comprehension**, **software visualisation**, **software optimisation** and **empirical software engineering**. Software visualisation is often used in research to aid program comprehension, thus those issues go in pair. However, research in software optimisation, and in particular for concurrent programs, is primarily focused on low-level engineering issues and is an area that has been under-explored from the human-computer interaction perspective.

In this chapter we selected, what we consider seminal and relevant papers which have allowed us to build towards a general framework that can be used to design various support tools for ordinary developers and aid them in the process of parallel performance problem diagnosis. The selection of the work presented in this chapter is relatively narrow and aligned closely with our research questions.

We first examine the domain of empirical software engineering which aims to understand and model the way programmers work; this includes the way they make sense of their code and software architecture, the metaphors, the way they test and debug the code or the differences between novices and experts. Understanding these things is important, as in our research we aim to support ordinary developers.

Next, we will look at the tools that programmers have at their disposal already. This includes algorithmic skeletons, taxonomies, performance prediction tools and a variety of visual analysis tools. Understanding how the existing tools function and how they can be used or improved is crucial for building effective support tools of any kind.

Finally, as we aim to create visual support tools, we need to understand the field of software visualization and performance visualisation in particular.

2.1 EMPIRICAL SOFTWARE ENGINEERING

While the practice of parallel programming for ordinary developers has not been widely studied from a HCI perspective, the scientific computing and software engineering communities have grappled with the problems associated with parallel programming for some time.

Much work has been carried out within the Software Engineering community with the aim of understanding and modelling the way programmers work and how their work-flows can be improved. One of the methodologies applied is to look at how complex strategies can be modelled in a series of simple observations, giving a better understanding of the daily practices of software developers and the architectural choices they face [87, 52, 14].

The practice of software engineering has been examined in various contexts, as the organisational environment can vary drastically. If one considers crowd-sourced software development, where a program is developed by a potentially unknown number of developers, in a distributed fashion, it presents challenges of task decomposition, coordination and planning [155], while video game development, typically conducted by a team of seasoned veterans under one roof presents another set of challenges and pipeline-like organisation [28]. Likewise, in multi-core software development, both the characteristics of the software being developed, and the development context will have an impact.

Over the past twenty years many studies have been carried out on novice programmers and identified the positive and negative aspects of today's programming systems [128]. Some of the research was closely related to program maintenance and can potentially be applied to performance analysis tools. While testing and debugging are two very complex areas for novice programmers, some researchers claim that programming tools should support source-level debugging with data visualisation to be more effective [22].

Using an appropriate concrete metaphor, a familiar analogy explaining how the programming system works, can have a positive effect on the usability [107] and also maximises transfer of knowledge if the metaphor is close to a real-world system [150].

Expert programming can be considered an opportunistic activity [58], however

some research indicates that expert programmers use intricate plans and strategies in order to schedule and prioritise their activities [13]. Planning is common among novice programmers and the absence of good planning strategies results in wrong assumptions and more bugs [21]. Programming tools should also consider locality; related components should be physically close [20, 153] and hidden dependencies reduce understanding [58].

For both novice and expert programmers, the ability to test partial solutions is an important feature [128]. Incremental running and testing, a programming strategy where the code is iteratively tested while new code is being written, has been found to be an effective debugging strategy [60], and it has been observed that developers perform better when such a strategy is adopted [56, 58, 131].

In addition, there have been attempts to understand how programmers work from a sense-making perspective, applying information foraging [133] to understand how developers debug [94, 132]. Various empirical studies and models have attempted to understand and help answer questions posed by developers [87, 52]. These studies have employed a variety of methodologies, including observations of pairs of programmers given sample tasks [148], and questionnaire-based studies [93, 122].

Prabhu et al. [134] present an extensive survey of software practices in computational science. They conclude that current programming systems and tools do not meet the requirements for scientific computing. They indicate that most tools assume that programmers would invest time and energy to learn and master a particular system, which turned out not to be the case with programmers wanting results immediately. They found that most scientists understand the importance of parallel programming in the context of scientific computing and many scientists spend a significant amount of time and energy programming. Despite this effort, most scientists seem to be unsatisfied with the performance of their programs and believe that the improvement of performance will significantly improve their research and, in some cases, allow larger experiments or enable fundamentally new research avenues. The survey concluded that overall the needs of computational scientists were under-served and new tools and techniques were needed to unlock the potential of high-performance computing.

In another study, Hannay et al. conducted an online survey of approximately two

thousand scientists in order to study software engineering practices in the scientific community [64]. Their study found that for "software testing" and "software verification" scientists assign on average a higher level of importance compared to the level of their own understanding of these concepts. Scientists seem to care a great deal about the correctness of their code, but feel they lack knowledge of software engineering practices to verify it. Additionally, their findings suggest that informal learning from peers was more important to scientists than formal education in an academic institution or formal training at the work place, they postulate that this may be due to the lack of appropriate formal training. Their study also confirmed the hypothesis that the majority of scientists use desktop computers most of the time for developing and use scientific software as opposed to supercomputers or clusters.

A classroom study conducted across a number of universities compared different programming models across a variety of representative applications [74]. It is worth noting that most of these studies use novices as the study participants rather than experienced programmers. The study compared shared memory (multi-core) and message passing (distributed) implementations of two problems, written by novice programmers. They found that the message passing development effort, measured in lines of code, was statistically greater but resulted in more correct programs. On the other hand, shared memory programs were smaller and easier to write but were more error-prone.

Luff conducted an experiment comparing developers' performance using various parallel programming models (actor model, transactional memory and standard shared memory threading with locks) while keeping the programming language and environment (IDE) the same [102]. The results were inconclusive and showed no significant difference in any objective measurement between those models.

A study conducted by Eccles et al. [45] had novice and expert programmers both categorising different parallel algorithms, using a card sorting method. They found that novices and experts used a different classification scheme: novice programmers organised problems around the problem domain while expert programmers organised problems around communication granularity and overhead. This difference in classification could identify a set of concepts which delineate novices from experts [145]. They concluded that the best way of organising parallel programming material

and libraries is around the expert classification scheme, and postulate that it would result in a more usable parallel software.

A wide variety of prior literature related to program maintenance focuses on debugging, with a particular emphasis on novice versus expert differences [110]. Most research agrees that reading and understanding code is the most common debugging method [43, 165], although this is not always feasible for very large programs [88]. Fix et al. [49] found that experts had more sophisticated mental models than novices, and so were able to use them more effectively to debug programs.

A study by Pancake [127] attempted to determine a correlation between mental models and the effectiveness of visualisations for parallel debuggers. It demonstrated that it is possible to implement various conceptual models using any programming language, however program development becomes significantly easier and more reliable when the language has support for expressing the desired model. The same correlation applies to the debugger visualisation models.

Fleming [144] conducted an exploratory think-aloud study in which he observed 15 programmers debugging a multi-threaded server application which was seeded with a defect. He claimed that the programmers who succeeded used a previously undocumented failure-trace strategy while debugging, and using such a strategy made the programmers more likely to succeed. The strategy involved modelling interactions between various threads in the program in order to find a failure trace (i.e. the interaction that led to a failure). He also postulated that cognitive strain may have been an important barrier as the failure-trace strategy was modelled internally. In addition, it is claimed that the inherent concurrency of parallel programming makes managing hypotheses regarding the cause of a bug more difficult.

2.2 PARALLEL PERFORMANCE ANALYSIS TOOLS

As will be argued later in the thesis, within parallel program development, the distinction between performance problems and bugs is much less clear than in traditional software development. However, the vast majority of parallel performance problems would receive only the broadest categorisation under existing taxonomies. Within Beizer's taxonomy [15] for example, in contrast to the detailed breakdown of other

categories of problem, the Performance category contains only *throughput inadequate*, *insufficient users*, *response time delay* and *performance parasites*, with the first three of these being more phenotype rather than genotype classifications.

When developers and researchers talk about parallel performance, they talk about it in the context of a particular algorithm, system or model. A multitude of effective design patterns have been recorded and studied in the literature, such as parallel for loops, concurrent containers, pipelines or map-reduce. These can be thought of as "algorithmic skeletons" [35, 162, 39, 6]. Such algorithmic skeletons can help reduce parallel programming errors as part of a "concurrency toolbox" with which programmers can construct the abstraction required to solve their problems and simplify the process of application development [25].

Recent years have seen widely accessible libraries providing various implementations of such skeletons becoming available, such as OpenMP, Microsoft Parallel Patterns Library (PPL), *java.util.concurrent* library or Intel Threading Building Blocks (TBB) [138, 143].

Those algorithmic skeletons represent general reusable solutions to commonly occurring problems within a particular context in software design - such solutions are known as *design patterns*. Design patterns are formalised best practices that the programmers use to solve common problems during the design phase. While the term *algorithmic skeletons* is in fact used for design patterns related to parallel programming, design patterns are widely used in other areas of computer science. Particularly, they have gained popularity in the field after the work of Gamma et al. [54] where some of the design patterns related to object-oriented programming were catalogued and explained.

In the Chapter 5 of this thesis, we present a model for parallel performance problem diagnosis, with the aim of supporting the design of effective performance analysis tools for parallel programming. Such tools can be seen as providing two types of capability - automation or performance prediction, intended to process the raw data and provide the developer with useful cues for action, and visual displays to be presented to the developer to support their own diagnosis and decision making.

A number of approaches to automatic performance prediction of parallel programs have been developed. For example, T. Fahringer in his recent book introduced novel

approaches to estimate various parameters that are critical for a well-performing parallel program, such as work distribution, computation time or cache misses [47]. Another example is combining user-selectable features for automated performance detection. This can be accomplished by using a hybrid system that allows a user to select a non-functional property (e.g. performance) and its features. For example, the performance of a database depends on whether a search index or encryption is used and how both features operate together, as the interaction of both features may lead to an unexpected behaviour while their individual presence may not [147]. Many other approaches exist and automatic performance prediction is an active field of research. However, accurately modelling and predicting performance becomes increasingly difficult for large-scale applications, since system complexity increases as well with its size [79, 173, 101].

Most bug/performance prediction algorithms have been developed, tested and verified in an academic setting. However, a recent case study by Lewis et al. [98] of a deployment of prediction algorithms within Google, concluded that while many developers are excited about having a new tool to help them in achieving better code performance, barriers remain in making them useful for developers. One of the main critiques of prediction algorithms is the lack of actionable messages, the presence of which might support wider adoption of automatic prediction tools. In addition, many performance problems occur only under specific input conditions, and automated profiled inputs do not generally cover all possible code paths [121, 81, 63, 177].

An important aspect of tool support for multi-core programming is understanding performance data. Given the volume and complexity of this data, visualisation is an important design direction as it leverages capacities and bandwidth of the visual system to quickly assess and understand large volumes of data. Visualising the behaviour of parallel programs is a very complex task, as the behaviour of the programs themselves is often complex. The area of effective visualisation techniques for parallel programs is still relatively unexplored within parallel programming research and has usability implications [145]. However, the need to form a scientific body of research, develop human-centered models, and target production level applications and their developers has been recognised [106].

Numerous tools to ease the engineering effort involved in the creation, debugging

and optimisation of parallel programs have been created, starting as early as the 1980's with the Poker environment [151, 152], allowing programmers to write and debug the first portable (cross-compiled) parallel programs.

Other tools, such as ParaGraph and ParaDyn have been developed to visualise behaviour of parallel software [65, 114]; most of the tools have been developed for the High-Performance Computing domain, and target distributed systems such as HPC clusters. While previous work has identified a number of broad issues and goals for tools to support programmers in understanding the performance of their programs, only a relatively small proportion of the literature deals specifically with the performance problems of multi-threaded programs.

With regard to tools, existing systems can be seen as providing answers to two main issues: between the 'topology' of software (e.g.: source-code hierarchy, memory layout, etc) and the mapping of such topology into the visualisation, as well as the issue of synchronisation. The topology issue requires that spatial relationships in programs be understood. The synchronisation issue requires various events occurring within the processor to be correlated [30]. While some existing visualisations are potentially useful, there is a need for analysis of how such tools can aid in the diagnosis of problems [12].

Many commercial tools, such as Intel® VTune, AMD® CodeAnalyst, Windows Performance Analyzer, GProf, IBM Rational PurifyPlus and others [5, 3, 7, 113] have incorporated performance analysis approaches that combine performance prediction metrics used in automated prediction and many different visual displays that present the information to end-user developers. However, according to a recent survey, a significant number of developers do not use any software tool at all: at least 25% for each type of tool surveyed, including memory and performance tuning tools. Significantly, 66% of the developers surveyed do not use any concurrency tool [1].

2.3 SOFTWARE VISUALISATION

It is important to understand the information visualisation field prior to designing a tool or advising information visualisation designers as we intended to do during

[1] The Parallel Programming Landscape: *http://www.danysoft.com/free/Intelparallelprog.pdf*

the project. This section is intended as a brief and general literature overview of the information visualisation field, since it is not possible to provide a comprehensive account of the field in merely a few pages. Below we describe the origins of the field and point out the most prominent collection of resources and continue with a few selected resources that relate to visualisations of performance of algorithms and programs within traditional and multi-core computing, including high performance and distributed computing.

In the data visualisation domain, there are two commonly accepted types of visualisations, namely explanatory visualisation and exploratory visualisation [99]:

- Explanatory visualisation consists of data visualisations that are used to transmit information or a point of view from the visualisation designer to its reader, they have a specific "story" that they intend to transmit.
- Exploratory visualisation is used by the designer for self-informative purposes to discover patterns, trends or sub-problems within the dataset.

We examine the field of information visualisation (InfoVis) and data analytics, as related to our research in parallel performance visualisation. InfoVis is an interdisciplinary field that has emerged from computing and graph-making, motivated by the need to visually represent increasingly large data-sets found in the sciences, as well as digital communications and records, to enhance how humans can analyse and learn from this information [100]. The field of data visualisation is relatively new and rapidly growing, driven by the latest developments in information and communications technologies, but tracing its origins to early mapping and graphing techniques [51, 163].

There are a number of books which aim to provide an overview of the new field of information visualisation and data analytics (e.g. [33, 83, 108, 154, 175]), along with numerous books for practitioners and designers (e.g. [99, 109, 116, 139]) and university courses around the world.

While there are numerous definitions of information visualisation, one of the most prominent is by Card et al. who state that InfoVis is "The use of computer-supported, interactive, visual representations of abstract data to amplify cognition" and also provides a useful survey of the origins of the domain [27]. Card et al. also identified

several streams of overlapping interest which concurrently contributed to the growth of the domain:

- Data graphics, which focused on the usage of graphs and maps to visually represent data, including setting guidelines and examples of good design [163, 164]
- Statistics and visualisation, which focused on the distilling and analysing multivariate and large datasets [34].
- Scientific visualisation, a research agenda on *visualisation and computer graphics* involving computational scientists and engineers, visualisation scientists and engineers, systems support personnel, artists, and cognitive scientists [111].
- User-interface research, which explored ways to help users to analyse large amounts of data, with a focus on usability and its cognitive amplification [27].
- Computer graphics and artificial intelligence, where researchers, informed by principles like those developed by Bertin and Tufte, sought to automate data transformation processes, as well as creating graphical and other visual representations [90].

Over the last decade and a half, numerous researchers have tried to apply some of the InfoVis concepts in order to visualise performance of various algorithms and systems. Early in 1993, Waheed and Rover described several issues that needed to be addressed to make performance visualisation of parallel systems possible and effective, namely: development of techniques into a concrete methodology for evaluating, optimising and predicting performance; rendering paradigms to display logical structure and behaviour; and integration methods in a performance analysis environment [167].

Unfortunately to date, most of the issues have been researched separately and are rarely brought together and evaluated as a whole. Performance prediction, for example, has made tremendous progress during the recent years, energised by a widespread adoption of multi-core processors [101, 173] and the performance, measured in cycles per instruction (CPI) can be predicted relatively accurately, even on complex server workloads [11] and different approaches to predicting are being proposed and improved [79]. Such research, however, has not been brought into commercial tools and has not been studied extensively from a user interaction perspective.

Some performance visualisation designs and considerations have been informed by the insights from research carried out on human factors, as mentioned earlier (e.g. [126, 65, 67, 66]).

Understanding software architecture is a crucial step towards building and maintaining software. However, software architecture is a conceptual and intangible entity, which can be difficult to comprehend and reason about; visual mappings can help to reduce the cognitive effort involved. The need of visualising the structure of a software system becomes particularly evident when the software system grows to entail a huge number of modules and procedures related in a complex fashion [55].

Research in software visualisation attempts to answer a multitude of questions posed by various stakeholders. In a world where most of the successful companies rely heavily on software, visualising software systems is important not only for software architects and engineers involved in its development but also testers, project managers and even customers.

Storey et al. [156] conducted a comprehensive analysis of cognitive models involved in the program comprehension process and describe a hierarchy of cognitive issues and their implications for design that should be considered during the design of software exploration tools, including software visualisation systems. Figure 2.1 illustrates the categorisation of cognitive design elements for software exploration. Amongst the different cognitive models, Storey et al. present two fundamental ones:

- **Bottom-up comprehension model** where the program understanding involves reading the program statements and constructing higher level abstractions.
- **Top-down comprehension model** requiring the domain knowledge of the program or previous exposure to the structure of the program. The maintainer (reader) of the program formulates a series of hypotheses and validates or rejects them by reading the source code or visualising the structure of the program.

In her paper Storey states that it is essential to determine which comprehension model (top-down, bottom-up or hybrid) is best supported by the tool, while some research also suggests that people are using either model depending on certain cues. The implications of this work have been extremely influential on the state of the recent research [125, 55, 105, 160]

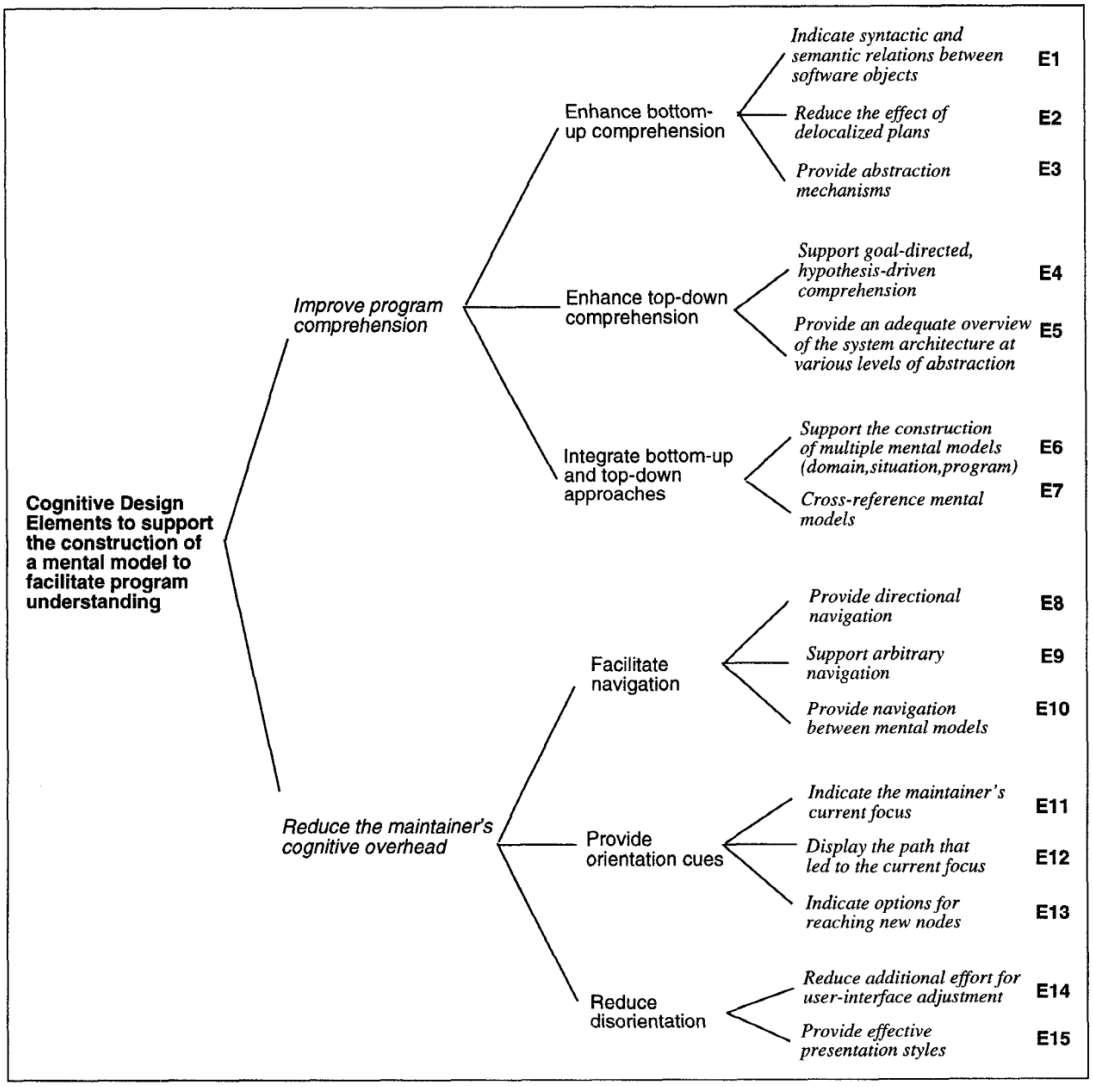

Figure 2.1 – The hierarchy of cognitive design elements for software exploration by Storey et al. [156]

Software itself is created, complex, abstract, and difficult to observe, to help programmers understand it, software visualisation uses various visual representations to make software more visible [32]. Roman and Cox [141] defined program visualisation as a mapping from programs to graphical representations. Advocates of visualisation point to the important role visual communication plays in our lives, to the very high bandwidth of the human visual system and our ability to detect visual patterns.

Several research surveys have been conducted in order to understand the current trends and the state of research related to software visualisation.

- A research survey performed by Koschke [89] on the use of software visualisation in the fields of software maintenance, reverse engineering and re-engineering

synthesised the work performed by 82 researchers. Koschke's survey showed that the vast majority of researchers believe that visualisation is very important to their domain. The survey also raises several concerns, including an improved understanding of the needs of viewers (programmers). Koschke calls for more modelling of visual understanding and experimentation with different kinds of visualisations. He also highlight that the most significant challenges in software visualisation arise in maintenance (debugging and optimisation), reverse engineering and re-engineering.

- A research survey of software software architecture visualisation conducted by Ghanam and Carpendale [55] highlights some of the trends in the research community. Such trends include the use of the third dimension in order to reduce the visual complexity. Another trend consists of exploring and using real-world metaphors (e.g. "cities") as means to amplify cognition. However, the authors call for experimental validation of various metaphors as there is little or no empirical evidence for the added benefits of metaphors in software architecture visualisation.

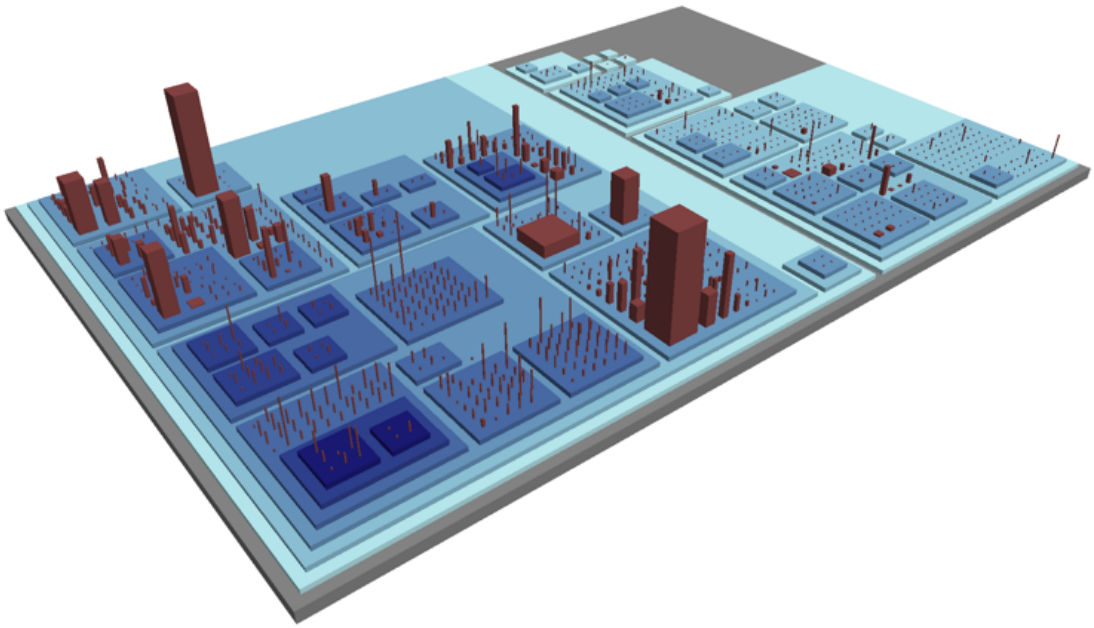

Figure 2.2 – Illustration of CodeCity, an integrated environment for software analysis. This tool represents various packages as city blocks and classes as city buildings and allows the user to map and visualise various software metrics such as complexity or length.

- A research survey conducted by T. Khan et al. [85] on the visualisation and evo-

lution of software architecture summarises the state of the research and various visual techniques used to display the software architecture hierarchies. It also highlights the importance of software metrics as an ideal abstraction as they encapsulate, summarise and provide essential quality information about the source code [92]. The survey highlights an essential aspect of software visualisation being the evolution of the software and some important works ([168], Figure 2.2) in the domain which combine software metrics, hierarchical structure, metaphors and effective temporal visualisation to gain a better understanding of the architecture and its evolution. Khan calls for forward collaboration between the researchers, experts and the industry, tailoring tools to meet specific requirements and conduct comprehensive evaluations of software visualisation systems.

2.4 EXISTING TOOLS

In this section we analyse some of the most widely used tools in the industry and review some of the advantages and disadvantages of each tool based on its capabilities or design language used. The analysis of the tools below is not intended to be comprehensive, but to be focused on on the subject of this thesis and the pros and contras of various tools represent advantages and disadvantages from the standpoint of ordinary programmers.

2.4.1 TOOLS FROM HARDWARE MANUFACTURERS

Hardware manufacturers such as Intel or AMD provide performance analysis tools that are closely-tied to their respective microprocessors. The tools make use of **hardware performance counters**, also known as hardware counters, which are a set of special-purpose registers built directly into the CPUs. They store hardware-related activities such as cache misses or power information. Some of the tools that fit into the category include:

- **Intel VTune Amplifier (Figure 2.3)**, a commercial application for software performance analysis created by Intel. This tool contains many advanced features that allow in-depth analysis of Intel-manufactured CPUs as it is able to collect

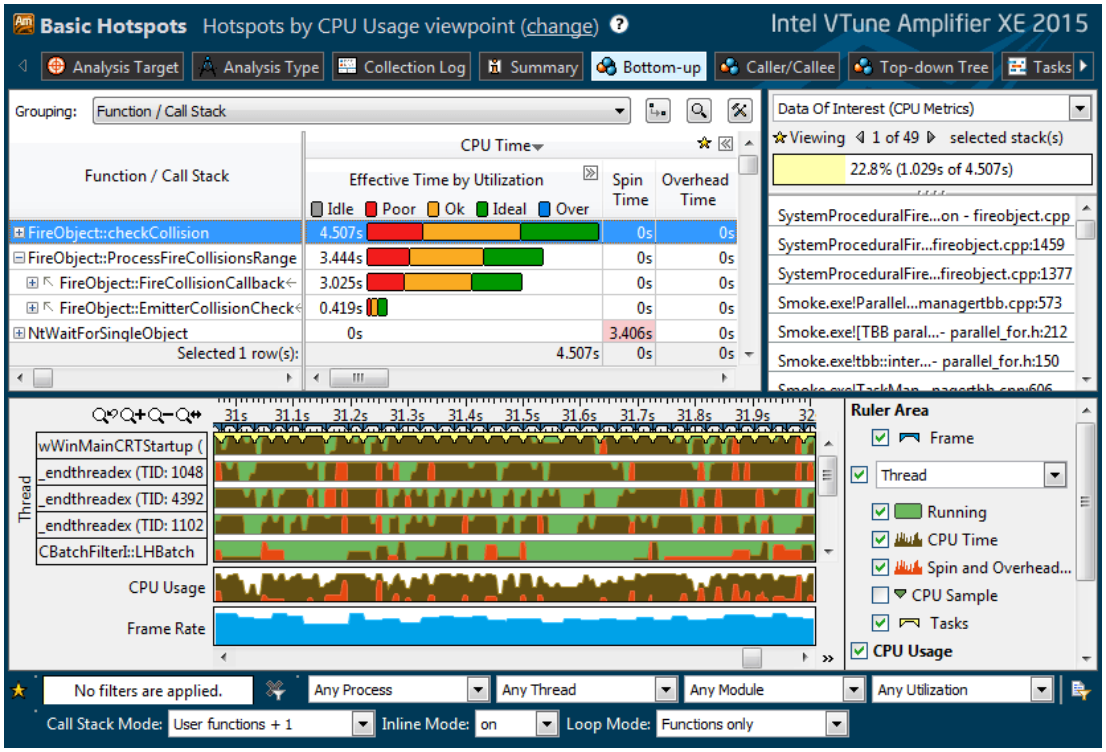

Figure 2.3 – Intel VTune Amplifier 2015

low-level information from the hardware performance counters present on the CPU itself.

- **AMD CodeXL (Figure 2.4)**, a software development tool suite created by AMD. The suite features a CPU profiler, both GPU debugger and a GPU profiler, along with a static OpenCL kernel analyser. The CPU profiler can be used to identify and improve performance of applications or drivers for AMD-manufactured CPUs. The profiler features instruction-based sampling and CPU hardware performance counters.

ADVANTAGES OF TOOLS FROM HARDWARE MANUFACTURERS

- Features hardware event sampling and allows to find specific tuning opportunities such as cache misses, branch mispredictions or even power measurement.
- Integrates expert knowledge of the engineers who have designed microprocessors. For example, VTune features various metrics based on low-level measurement, for example, the tool has integrated formulas which allow it to assess the

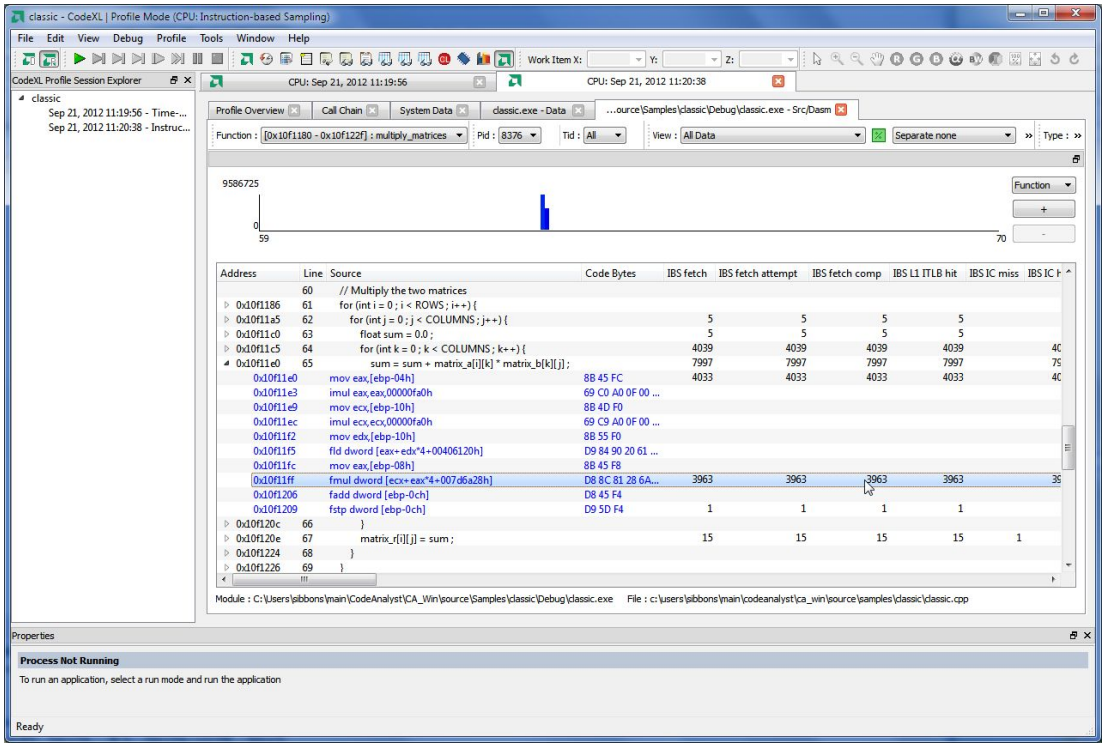

Figure 2.4 – AMD CodeXL

performance impact of cache misses at a particular level, as measured by an approximate number of the CPU cycles wasted due to the cache misses.

- Often have simple timeline visualisations that allow identification of load balancing and synchronisation issues.

DISADVANTAGES OF TOOLS FROM HARDWARE MANUFACTURERS

- At least some of the advanced features are not present on other platforms due to hardware incompatibilities and the support of such hardware is limited.
- Expert-oriented tools which require extensive training and manufacturer-specific vocabulary (eg: Intel QPI, AMD ISB).
- Visualisations seem to be a second-class citizen and built on-top of existing performance analysis metrics.

2.4.2 GENERIC OPERATING SYSTEM TOOLS

Operating system vendors such as Microsoft or various third parties provide generic trace analysis tools which take operating system events or various log files and allow interactive analysis or querying of the information contained in the log files. One of the tools that fits into this category is **Microsoft Windows Performance Analyzer (Figure 2.5)**, a tool that creates graphs and interactive data tables based on the data provided by the underlying Windows subsystem, namely Event Tracing for Windows (ETW). This tool is generic in design and can plot any time-series of precise or sampled events provided by the subsystem. It features deep integration with Windows platform, querying and comparative analysis visualisations.

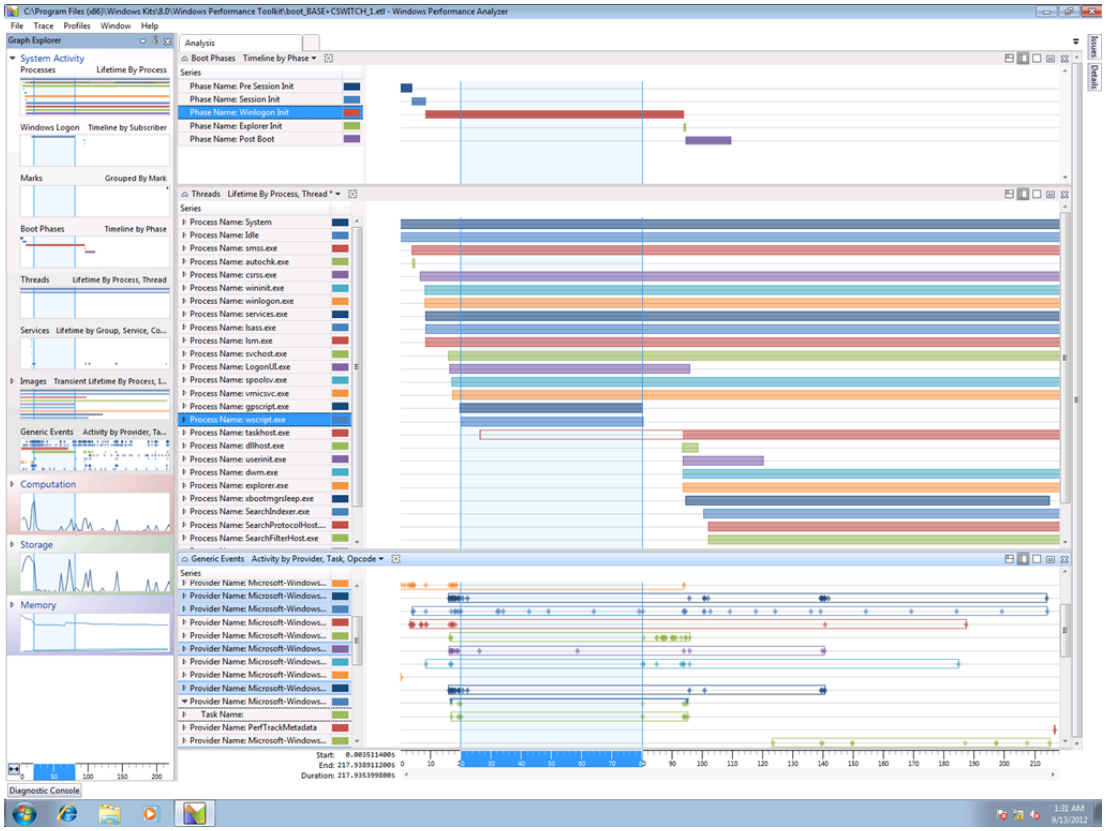

Figure 2.5 – Microsoft Windows Performance Analyzer

ADVANTAGES OF GENERIC OPERATING SYSTEM TOOLS

- Rely on an external and standardised data collection mechanisms. For example in Windows, such a mechanism is built-in to the Windows Kernel and al-

lows tracing of operating system events with a single command, which is very straightforward for novices to get started.

- Designed to support the process of data exploration and allow generation of common time-series graphs such as line charts, timelines or area charts.
- Have standard features for exploring large datasets such as zooming. For example Windows Performance Analyzer allows row/column-based brushing and linking with visualisation and a data table dynamically linked together.

DISADVANTAGES OF GENERIC OPERATING SYSTEM TOOLS

- Generic in nature, such tools do not provide generic visualisations or specific expert knowledge on the visualisations as it allows advanced analysis of any kind of performance data.
- Operating system event tracing mechanisms do not usually provide extensive sampling of hardware performance counters at the date of writing this analysis, which makes it impossible to identify lower-level performance problems such as poor data locality or false sharing.

2.4.3 CONCURRENCY VISUALISATION TOOLS

This category represents tools designed specifically for the purpose of concurrency or multi-threaded analysis, be it for debugging or performance analysis purposes. Some of the tools that fit into the category include:

- **Microsoft's Concurrency Visualizer (Figure 2.6)**, an optional extension to Microsoft Visual Studio 2013 and later, the Microsoft's flagship integrated development editor (IDE). The Concurrency Visualizer uses the ETW subsystem, similar to the WPA described above, however the tool is designed to help programmers examine the performance of multi-threaded applications. The tool provides graphical, tabular and textual data to depict the relationships between the threads in the application and the system as a whole. The Concurrency Visualizer can be used to locate, amongst others: performance bottlenecks, CPU underutilisation, thread contention, cross-core thread migration, synchronization delays, DirectX activity, areas of overlapped I/O.

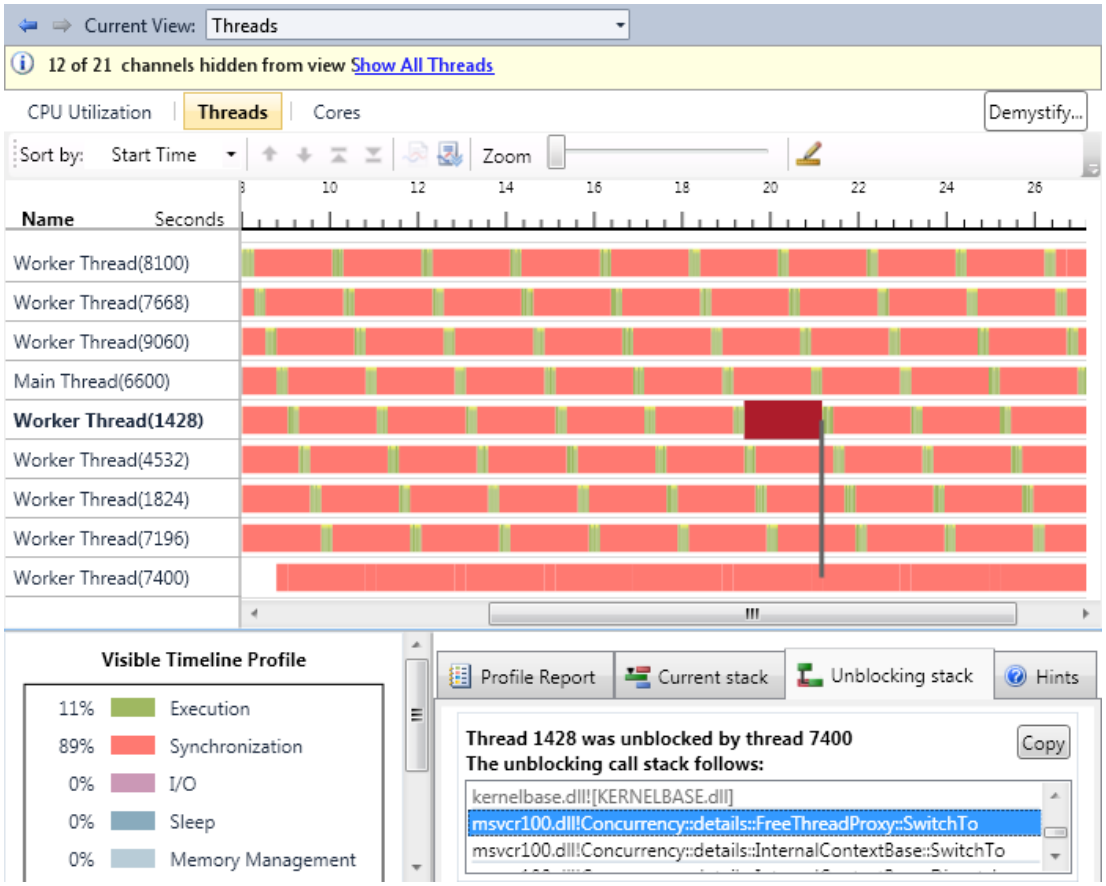

Figure 2.6 – Microsoft Visual Studio Concurrency Visualizer

ADVANTAGES OF CONCURRENCY VISUALISATION TOOLS

- Embeds some expert knowledge and does further analysis. For example, the Concurrency Visualizer logically splits the execution time in different components such as synchronisation or memory management.
- The tools in this category often include a collection of typical and recognisable visual patterns that are exposed through it, together with an explanation of the behaviour that is represented by each pattern, the likely result of that behaviour, and the most common approach to resolve it. The collection includes patterns such as lock contention, uneven workload distribution, oversubscription and inefficient I/O.

DISADVANTAGES OF CONCURRENCY VISUALISATION TOOLS

- Operating systems event tracing mechanisms do not usually provide extensive

sampling of hardware performance counters, which makes it impossible to identify lower-level performance problems such as poor data locality or false sharing.

2.4.4 PLATFORM-SPECIFIC PROFILERS

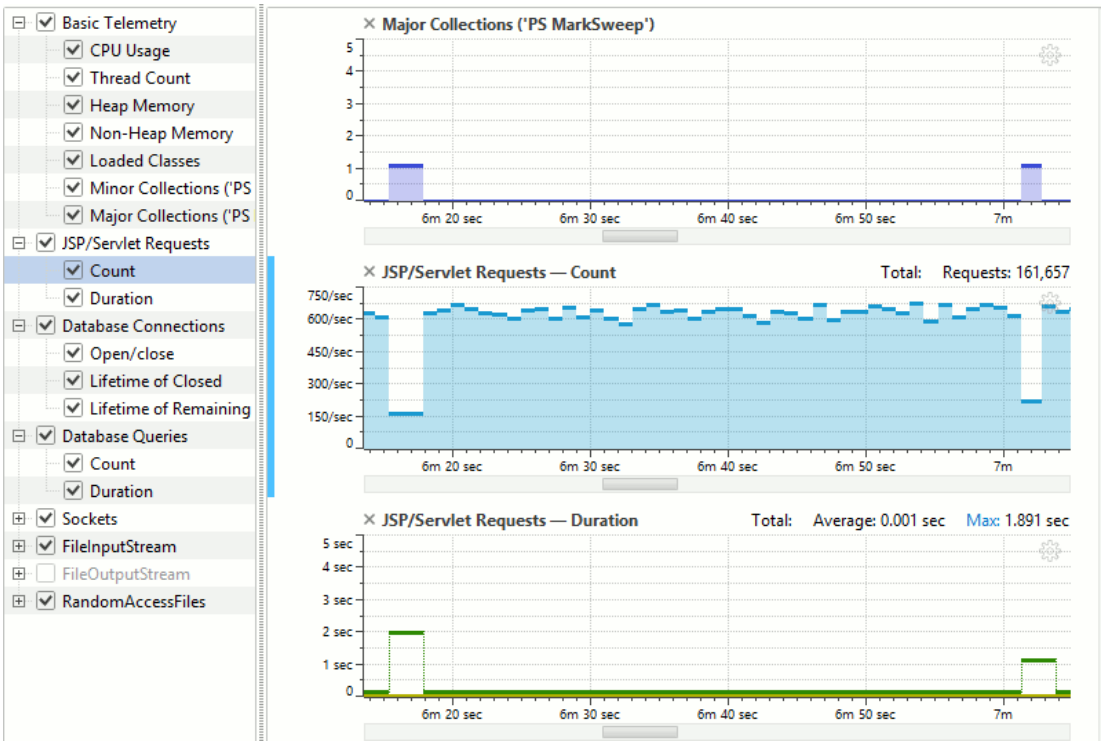

Figure 2.7 – YourKit Java Profiler

Many languages and platforms such as Java or .NET have an ecosystem of tools specifically designed to examine the performance of the applications running on those platforms. In the case of Java and .NET, such tools often have access to the underlying Virtual Machine (VM) and garbage collector and provide platform-specific diagnostic capabilities. Some of the tools that fit into the category include:

- **YourKit Java Profiler (Figure 2.7)**, both a CPU and a Memory profiler developed by YourKit. The tool exists in two versions, one targeting Java and another is targeting .NET platforms. Each version has various platform-specific features such as Java Servlet Page (JSP) requests graphing or Java VM statistics.
- **ANTS Performance Profiler (Figure 2.8)**, a CPU profiler developed by Redgate. The tool is targeting specifically .NET platform and related concepts and collects

statistics from the .NET virtual machine along with some Windows performance counters.

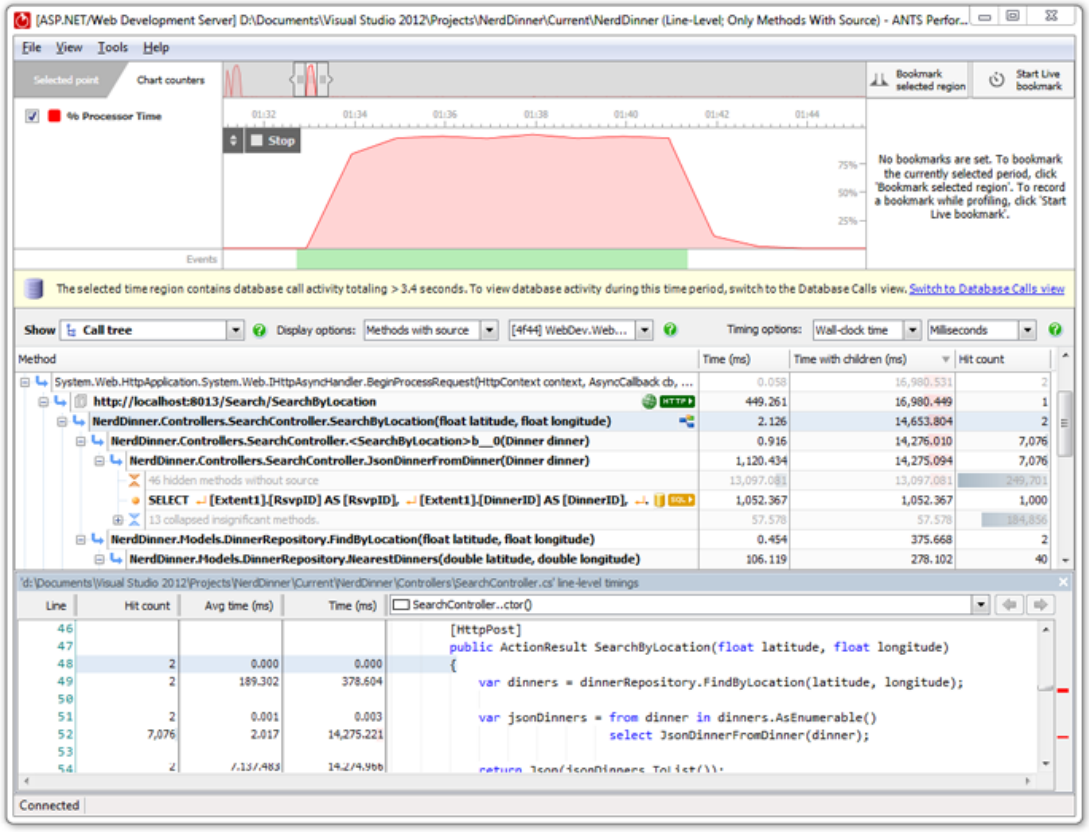

Figure 2.8 – Redgate ANTS Performance Profiler

ADVANTAGES OF PLATFORM-SPECIFIC PROFILERS

- Often features a call tree visualisation which is widely-used in industry and included in many profilers, such as AMD CodeXL described above.The call tree depicts the performance data for every method and is usually enhanced with sparkline visualisations to allow quick identification of hot spots.

- Relatively easy to use by programmers experienced in the particular language or platform, in the case of .NET it uses the same vocabulary of concepts (eg: ".NET GC Gen0").

- Since the tool targets a specific platform, it is able to show additional semantic information about various libraries and group them. For example, the ANTS

Performance Profiler can show "Database calls" depicted in a tabular fashion and sorted by the execution time.

DISADVANTAGES OF PLATFORM-SPECIFIC PROFILERS

- The embedded visualisations are of limited use for concurrency purposes. For example Redgate ANTS features a simple timeline visualisation of the aggregate CPU usage across all cores and threads.

2.4.5 HYBRID CPU/GPU PROFILERS

In recent years, with the advent of general purpose computing on graphics processing units (GPGPU), graphic hardware manufacturers began to provide hybrid performance analysis tools for their clients, often game companies. Such tools integrate CPU and GPU profiling techniques and allow side-by-side performance diagnosis. Some of the tools that fit into the category include:

- **NVIDIA Nsight (Figure 2.9)**, an extension of Visual Studio or Eclipse integrated development environments (IDEs) containing a suite of tools that brings CPU and GPU debugging closer to the place where the program is actually being developed (the IDE). It features timeline visualisation for hybrid profiling of both CPU and GPU activity, along with support for GPGPU (CUDA), shader programming and real-time inspection of Direct3D API calls.
- **AMD CodeXL (Figure 2.4)**, a software development tool suite created by AMD. The suite features a CPU profiler, both GPU debugger and a GPU profiler, along with a static OpenCL kernel analyser.

ADVANTAGES OF HYBRID CPU/GPU PROFILERS

- Provides a unified visualisation of the CPU and GPU application activity, allowing the identification of performance bottlenecks that can occur during CPU/GPU interaction. For example, this would allow identifying application slowdowns occurring due to data (eg: textures) being transferred to the GPU memory from the main memory.

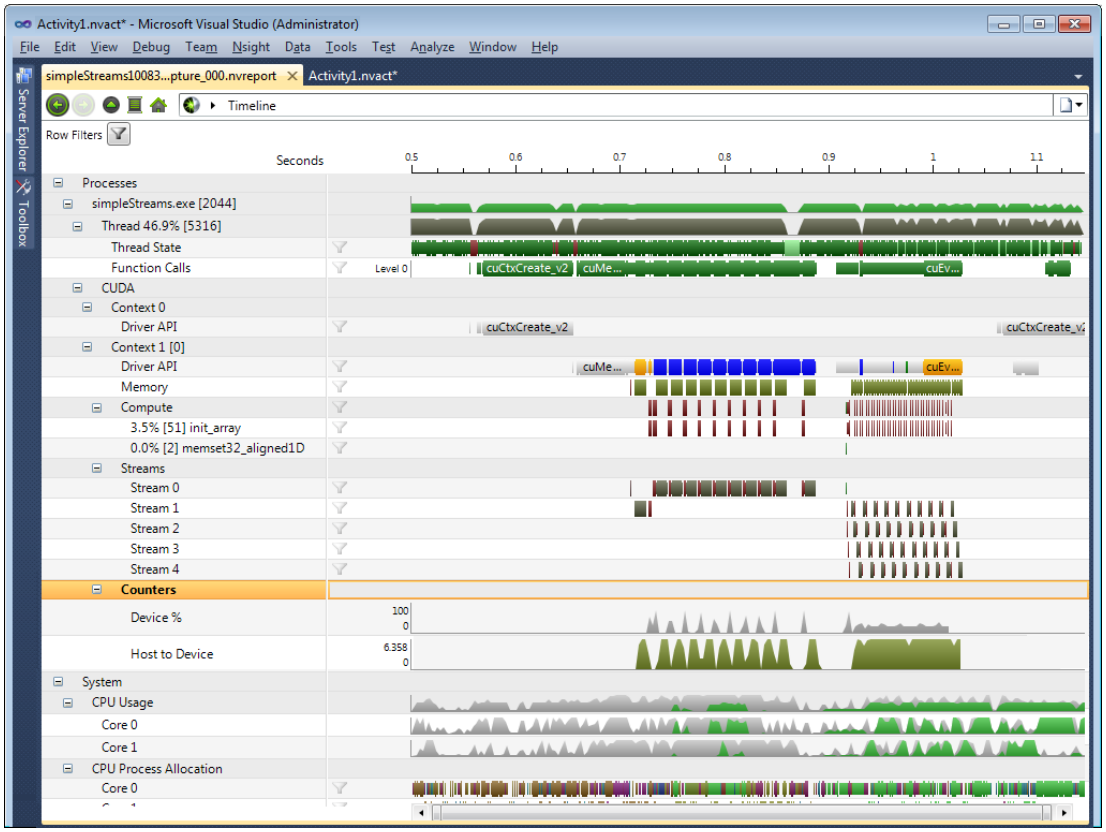

Figure 2.9 – NVIDIA Nsight

- Supports a range of time-series visualisations and logical separations such as streams, threads, CPU cores or memory.

DISADVANTAGES OF HYBRID CPU/GPU PROFILERS

- Targets primarily special-purpose hardware (e.g.: NVIDIA) as well as game developers. The use for ordinary application developers is limited as they would use higher-level frameworks instead of OpenGL or DirectX directly.

2.5 CONCLUDING REMARKS

The last decade has seen an increase in available performance analysis tools, coming from both academia and industry. This trend is driven from multiple directions simultaneously. The rapid growth of distributed computing with the advent of the cloud is pushing the need for better HPC performance diagnosis tools such as TAU [146] or PerfExplorer [76]. The increased diversity or parallel processors and the number of

cores on commodity hardware have pushed the creation of hardware-specific analysis tools such as Intel® VTune Amplifier [137] and AMD® CodeAnalyst [44] which leverage hardware performance counters to achieve "closer to metal" performance diagnosis.

The increased prevalence of parallel hardware has also influenced developers specialising in popular programming environments such Java or .NET who require support for parallelising and optimising their systems. To support them and leverage platform-specific constructs such as garbage collectors or just-in-time runtime diagnosis, specialised tools have been created such as Jinsight [38], HProf, XProf, JProfile or YourKit [118] to name a few.

At the same time, significant effort within popular operating systems such as Linux or Windows have pushed for better support of parallel hardware and the need for better performance diagnosis of concurrently running processes. One examples is the Microsoft Windows Performance Analyzer, based on the event tracing subsystem within Windows operating system (ETW) [130, 117] which allows visual analysis of every single process/thread running on a particular machine.

While the need for better performance diagnosis tools is undeniable and highly driven, there has been little work related to the more general analysis of the problem space to serve as a solid scientific foundation consisting of models and actionable insights to support the development of the performance tools. In this thesis, we aim to remedy the situation by providing such a foundation; or at the very least a good starting point for one.

The empirical software engineering community have studied extensively the processes that relate to the construction of software. We can draw on the methods commonly used for better understanding the daily practices of software developers with regard to the process of performance tuning of parallel programs. Understanding the similarities between practices involved in the process of debugging and optimisation of parallel software might bring us a step closer to applying existing models to the domain of our interest.

Numerous algorithmic skeletons have been developed in the recent years with the aim of making the creation of parallel programs more accessible to ordinary developers. They simplify the application development, reduce errors and potentially reduce

mental effort required. If we can understand some of the reasons for the success of the algorithmic skeletons, this might inform us of the way programmers think about creating parallel programs and help us to support them in the process of optimisation.

Finally, such a foundation should also provide a model or insights to simplify the creation or improvement of performance analysis tools. A visual tool can be useful here, given the volume and complexity of the data that can be collected through the means of underlying hardware, operating system or program instrumentation; some of the early successes over the last decade and a half in applying InfoVis concepts to performance visualisation along with existing tools that support of some kind of visualisation, suggest that this is a promising direction for research.

However, these issues must be studied in relation to one another and brought together and evaluated as a whole.

CHAPTER 3 RESEARCH OVERVIEW

The goal of the research contained within this thesis is to support the creation of software development tools that allow developers to understand the performance of software on many-core systems. Such tools must support the developer in understanding the complex interactions of software with dozens of threads running on multi-threaded systems. This entrails the development of summaries and visualisations that help to understand the patterns of parallelism in multi-threaded software for software maintenance and moving to systems with more cores. The main objectives are thus:

- to provide a foundation for the development of software tools that will help the developer to understand the complex interactions of software with dozens of threads running on many-core systems by building an understanding of the problem space;
- to explore the solution space for development of summaries and visualisations that help to understand the patterns of parallelism in many-core software for software maintenance and moving to systems with more cores.

In this chapter we discuss some of the high-level research processes and the research methodology we have applied to answer the research questions and accomplish the goals set out above.

This research combines a number of different research approaches and hence follows a mixed **mixed methods research methodology** [37, 80]. The process is illustrated in the Figure 3.1 and follows the basic exploratory sequential design of mixed methods research, in which we have collected and analysed both qualitative and quantitative data in response to our research questions. Each step of the process was informed by the insights, results gathered in the previous steps and established qualitative and quantitative methods.

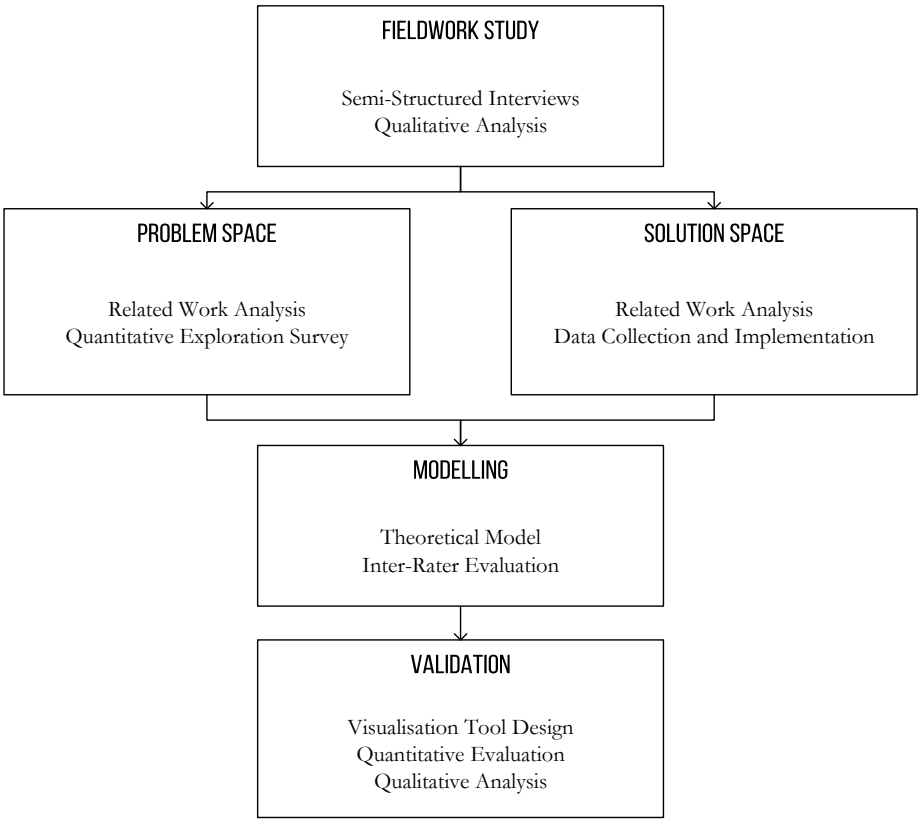

Figure 3.1 – An overview of the research process.

The main reasons for using a mixed methods research methodology are the complexity of the field and the relatively under-explored nature of performance tuning of parallel programming. To effectively establish a scientific foundation for tool designers, we found ourselves in need of employing diverse methods to explore, model and validate various elements of our research. These elements then had to be brought together and evaluated as a whole through the proxy of a visual tool; a convergent mixed methods design was used to accomplish the latter.

Mixed methods research methodology was used as we needed to augment the qualitative data gathered through various interviews and surveys with statistical data and a more rigorous validation. Combining both statistics and qualitative information gave us a more complete picture of the research problem and therefore, allowed us to build a more robust scientific framework for building tools to support parallel performance optimisation process.

While this methodology was extremely helpful and allowed us to build towards such a framework, it had some disadvantages and challenges associated. One of the

main challenges for us was to find a way to effectively mix the results of different methods in order to achieve a result which is bigger than just the sum of two parts. For example, as will be explained in the Chapter 5 we extended on the results of our qualitative interview analysis and taxonomy construction and created a model that relies quantitative inter-rater agreement calculation in order to figure out which parts of our taxonomy can be applied "as is" and which ones need further work. Another example of such mixing is our validation study where we have evaluated our visualisation tool and discovered different interesting aspects in our data with different methods, effectively one complementing the other. Another challenge is that the process of merging qualitative and quantitative data is very time-consuming and complex, as it requires analysing, coding and integrating data from unstructured to structured data [42].

3.1 RESEARCH QUESTIONS

We have mentioned in the Chapter 1 the three main research questions, which fundamentally guide the process of the research presented in this dissertation.

- **RQ 1: How do people conceptualise parallel programs?** We try to answer this question by conducting a series of informal and unstructured interviews with a broad range of developers across industry and academia and understand the conceptual models they employ while designing well performing parallel programs and the various implications. The interviews are analysed and distilled as a series of practical implications for design, that can be leveraged by both tool designers directly and by us for further modelling of the problem space.

- **RQ 2: Can we perform a comprehensive and systematic analysis of the information involved in understanding and improving the performance of parallel programs?** We try to answer this question by designing a taxonomy and model, based on the findings from the interviews and creating an effective visualisation metaphors to match those models. To this end, we employ a combination of both quantitative and qualitative analysis methods to construct practical models that can be used later on to guide the design and implementation of the tools to support programmers of parallel systems.

- **RQ 3: To what extent can a visualisation effectively support programmers in the task of optimising parallel programs?** We try to answer this question by creating a prototype of a particular parallel performance problem, in our case this being "data locality problem", one problem category from the taxonomy, and then evaluating those prototypes both quantitatively for the correctness of problem identification and qualitatively so we can understand better the process undertaken by programmers to solve some of the particular tasks related to data locality problems.

3.2 UNDERSTANDING THE PROGRAMMER

We began our research by conducting a series of unstructured interviews with programmers across industry and academia in order to understand the conceptual models they employ while designing well performing parallel programs and the various implications. The interview results and the study itself are discussed in more detail in the Chapter 4.

In our interview analysis we have performed both open and axial coding for the entire set of transcribed interviews. The codes were then grouped into logical categories such as techniques, people or tools and then grouped into abstract sub-categories. Figure 3.2 depicts the entire hierarchy that emerged during the interviews. We have also made use of interactive visualisations to explore the hierarchy and drill down to individual statements.

During the entire process of analysing and annotating we produced a large number of different memos[1], which were used to perform a deeper analysis than just a categorisation, we used them extensively in order to derive the implications for design for tool builders regarding how modern tools that support regular programmers in their endeavour of parallel performance optimisation should be built. In the process of analysing the interviews, we also constructed some interactive web-based visualisations to aid us in the task of exploration and in order to better understand the interview transcripts.

One of our main findings is the importance of **orchestration models** in the domain

[1] Memos are various snippets of text and audio we recorded along the way, representing our thoughts and ideas related to the interviews, as they emerge during coding, data collection, analysis and memoing.

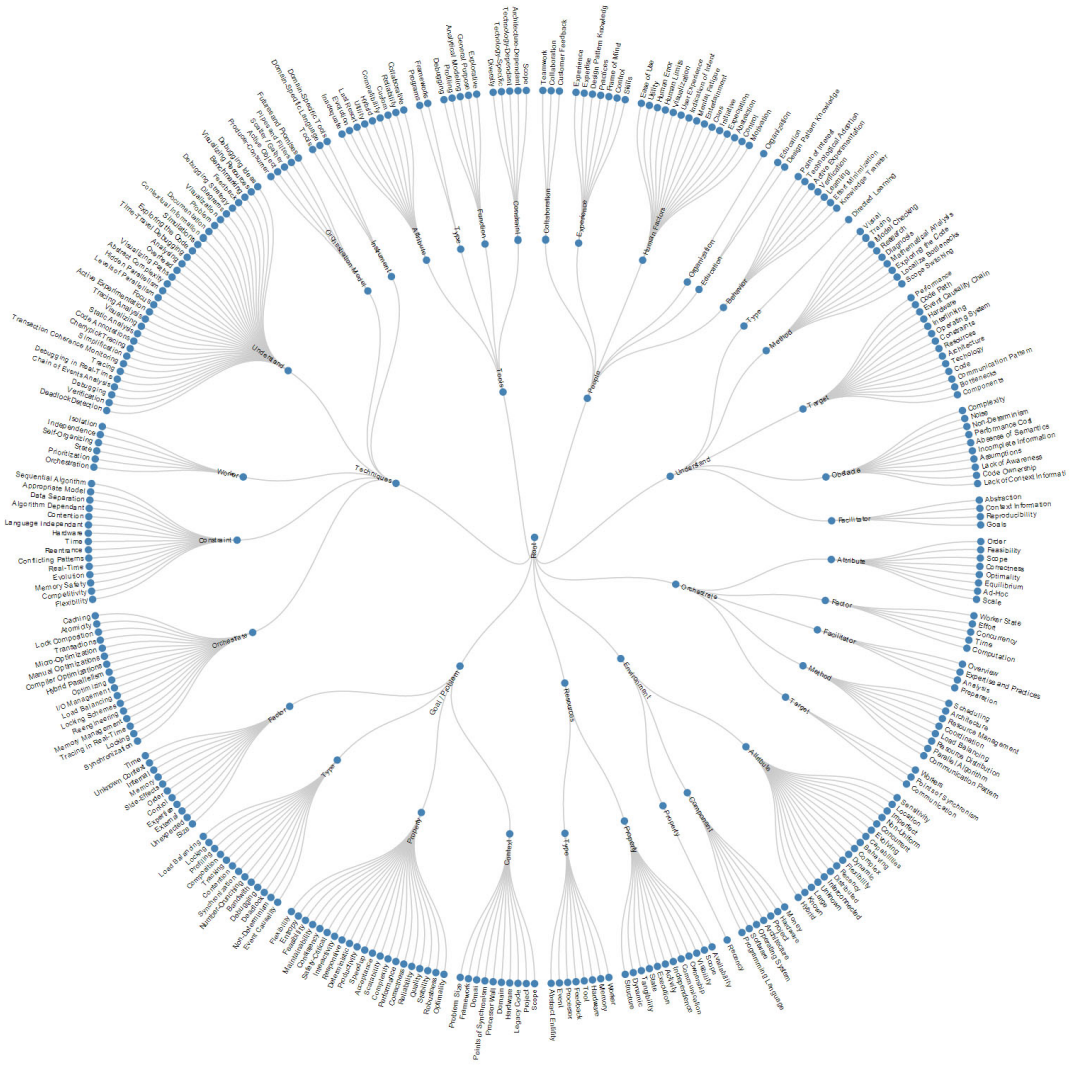

Figure 3.2 – The radial tree representing all the categories, sub-categories and individual codes that emerged during the analysis of the interviews.

of parallel programming. An orchestration model is a specific design pattern manifested by the way programmers arrange, coordinate and manage a set of workers using and sharing resources in order to achieve a common goal. Orchestration models are abstract design patterns, spanning different contexts and patterns related to software, hardware and system architectures. An easy metaphor to understand this, is to consider that a developer who writes parallel programs does not write recipes (as often taught to first year computer science students), but manages a project with several workers working for him. The developer attempts to orchestrate these workers in an optimal way, considering the available resources and tasks at hand. In the parallel pro-

gramming context, such workers can be threads, processes, machines, domain-specific classes, etc.

Orchestration models are a type of design pattern employed in parallel programming, and they span different contexts (or scopes) on which developers are focusing. In software engineering, design patterns are often related to reusable code skeletons for quick and reliable development of parallel applications [149]. In contrast, orchestration models are more abstract, spanning not only patterns related to software architecture, but hardware and system architectures as well.

3.3 MODELLING THE DIAGNOSIS

In the second phase of our research, which will be explained in more detail in Chapters 5 and 6, we wanted to answer the question of whether we can perform a comprehensive and systematic analysis of information involved in understanding and improving the performance of parallel programs. In other words, we needed to create a **practical model** that could be used to build tools to support developers. To do this and continue with the goal of creating an effective visualisation for parallel problem identification, several parts had to be brought together and analysed as a whole:

1. **Performance data extraction and collection**. In order to be able to build parallel performance analysis tools, we first need to be able to collect data whether it be by instrumenting the program or gathering operating-system or hardware counter data. Most importantly, we need to know what type of data is possible to collect, visualise and analyse later. We settled on the Windows operating system, since it features a set of well-supported and well-documented tools. We compiled comprehensive lists of of different measurable events or counters we could collect; this gave us insight into the possible information that can be displayed by tools that would allow successful identification of performance problems.

   Once this was carried out, we implemented a data collection tool that allowed us to experiment, extract and combine both CPU hardware and performance counters[2] and performance events and counters issued by Windows operating

[2]Hardware performance counters are a set of special-purpose registers built into modern micropro-

system. Chapter 6 contains more discussion on this aspect of our research.

2. **Taxonomy of parallel performance problems on multi-core architectures**. Unfortunately, no comprehensive taxonomy existed, listing parallel performance problems which can occur on shared memory multi-core architectures; hence, we had to create one ourselves. Several classes of problems had been identified tangentially in the interview study, and we began iteratively constructing a taxonomy based on various scientific literature and white-paper publications from companies such as Microsoft or Intel. The very first taxonomy contained only eight of the most common problems, or at least that was our untested initial assumption. The initial problems included under-subscription, over-subscription, uneven load distribution, lock contentions and lock convoys, along with I/O contention, indirect memory access and false/true sharing.

   After the first model we reiterated multiple times with two domain experts and created a more complete taxonomy which contained seven broad categories such as load balancing and task granularity, with a total of twenty three individual problems. We then performed a broad survey with 71 participants to better understand which problems are most commonly occuring as well as commonly diagnosed, which ones are more exotic and do not really occur in practice. We discuss this in more detail in the Chapter 5.

3. **Expert knowledge on the parallel problem diagnosis**.

   By the time we started creating the taxonomy, we had already begun pulling in a significant amount of expert knowledge. Together with the domain expert, we have attempted to create a set of simple diagnosis models for the performance problems we have identified. Some of the diagnosis process turned out to be rather straightforward while other problems turned out to be very difficult to diagnose and the expert could not determine how he would perform diagnosis. For example, for one of the **poor data locality** performance problems we were able to construct a simple decision tree based on the observations of measurable events or counters we could collect.

cessors to store the counts of hardware-related activities within computer systems. Advanced users often rely on those counters to conduct low-level performance analysis or tuning.

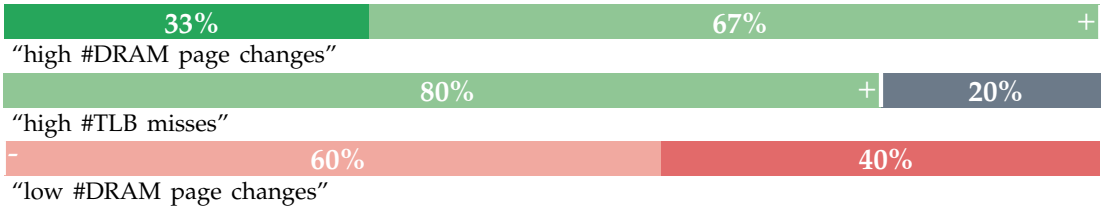

Figure 3.3 – An example of the levels of agreement between experts on various "measurable observations" of two performance problems.

Once we had completed the three parts, we knew the data we could collect and implemented a performance data collection mechanism. Once we had a relatively large and refined problem taxonomy and initial expert diagnosis models for most of the performance problems, we began to work on a **practical model** of performance problem identification. It was paramount that the model would be relatively easy to apply as the model is intended to be used by practitioners and tool builders and not only for research purposes.

While Chapter 5 goes into more detail on the model itself and the related validation, an example of the components of the model can be seen in Figure 3.3 which depicts the levels of agreement between experts on various observations that can be either:

- **A (strong) indication** of a particular performance problem being present in the target program.
- **A (strong) contra-indication** of a particular performance problem being present in the target program.

In other words, this model helps to determine which measurable events or counters can be used for effective parallel performance problem identification through inter-expert validation and, can be extended by simply having experts assessing various observations and the agreement level statistics used to further extend/refine the model.

3.4 VISUALISING THE PERFORMANCE

The third, and the last phase of our research consisted in applying our own findings and models we have established during previous phases and validate them by creat-

ing an effective visualisation to effectively identify several parallel performance problems. We decided to address data locality issue and use our observational model along with the diagnosis model for the design.

The visualisation we have designed consists of three main components and is explained in detail in Chapter 7; the main design goal of the tool is to allow effective identification of data locality issues. In the past two decades processing speeds increased by around 50% per annum, whereas the time to access DRAM memory fell by only 10%-15% per year. The result is that it now takes hundreds of processor cycles to read a value from main memory. This phenomenon is often called the "memory wall" and efficient use of CPU caches is required to achieve good performance; this recent trend influenced our decision to address data locality problem in our visualisation.

Once we had a visualisation, an evaluation was performed using a convergent basic design as part of our mixed methods research, to assess whether programmers could identify correctly data locality problems. The visualisation was compared to a simple source-code reading exercise, since there is a large percentage of programmers (+- 60% as found by one of the studies sponsored by Intel) who do not use any parallel performance tools and analyse their performance problems by simply looking at the source code. This was consistent with the feedback we received during the interviews. In our experiment we have found, among other things, that the correctness increased significantly across the board (experts and non-experts alike), along with an overall confidence boost. Chapter 7 goes into more detail on the experiment and the findings.

CHAPTER 4 UNDERSTANDING THE PROGRAMMER

A qualitative study was carried out to better understand how developers approach parallel programming, identify the issues that they are trying to address, and how software performance analytics systems could help them in their work. We conducted a range of interviews across various organisations including a large corporation producing operating systems, one of the largest B2B software corporations, universities, research laboratories and small to medium sized software companies. A broad spectrum of organisations was targeted in order to obtain a general overview of the field of parallel programming practices and problems.

4.1 METHODOLOGY

Interview participants were practicing software developers, engineers and academics. All of the interviewees were practicing parallel programming in some way, including high performance computing (HPC), Graphics Processing Unit (GPU) accelerators, many-core and/or multi-core programming. The goal of the semi-structured interviews was to explore and understand the daily activities and challenges faced by programmers and to explore the techniques they use in order to tackle those challenges. The interviewees were asked to describe the nature of their work and recent projects involving parallel programming.

Overall, 22 people were interviewed. 14 of them were recorded, resulting in over 8 hours of audio. Interviews were transcribed (around 47,000 words), open coded (582 open codes, unified into 252 codes), categorised in 8 major categories and sub-categorised in 23 sub-categories. Interview participants are summarised in Table 4.1.

Interviews were semi-structured, and interviewees were asked to talk about the challenges they had to overcome in their every day work related to parallel program-

Participant	Organisation	Activity	Years
P1, Male	Corporate	Engineering	0-5
P2, Male	Corporate	Engineering	0-5
P3, Female	Corporate	Engineering	5-10
P4, Male	Corporate	Engineering	5-10
P5, Male	Corporate	Engineering	10+
P6, Male	Corporate	Engineering	10+
P7, Male	Corporate	Engineering	10+
P8, Male	Corporate	Engineering	10+
P9, Male	Corporate	Engineering	10+
P10, Male	Corporate	Research	10+
P11, Male	Corporate	Research	10+
P12, Male	Corporate	Research	10+
P13, Male	SME	Consultancy	5-10
P14, Male	SME	Consultancy	5-10
P15, Male	SME	Consultancy	10+
P16, Male	SME	Engineering	5-10
P17, Female	SME	Engineering	10+
P18, Male	SME	Engineering	10+
P19, Male	University	Research	0-5
P20, Male	University	Research	5-10
P21, Male	University	Teaching	5-10
P22, Male	University	Teaching	5-10

Table 4.1 – A table of participants with their years of experience, main activity and the type of organisation.

ming, the tools they had employed, and practices they used. While this analysis method has its roots in Grounded Theory [157], we would like to highlight that it was not our goal to build a holistic theory from the data. Following common practice in HCI [115], we used the approach as the foundation for a focussed analysis of the transcribed interviews.

Listed below are the some of the questions that we asked developers during our interviews. In keeping with the methodological background, the questions were deliberately kept general and we tried to "let them talk" from these starting points:

- Could you introduce yourself, describe your background and professional experience?
- Could you describe some projects you worked on and what kind of challenges you have met in these projects?
- What kind of techniques do you use to optimise your programs?

- What kind of interesting projects have you had and how did you solve them?
- What kind of challenges, what kind of problems do you face on a daily basis?
- When you've got a piece of code, how do you find where it should be optimised?
- What kind of profiling, tracing tools are you using?
- How do you usually determine where the bottlenecks are in the code?
- You spoke about knowledge of hardware. Could you specify what kind of knowledge of hardware you needed?
- You mentioned patterns in parallel programming, do you use them? Are they relevant?
- Alright, do you have something else to add?

4.2 INTERVIEW ANALYSIS PROCESS

As mentioned before, the process for the interview analysis has its roots in Grounded Theory, and this section explains the analysis process we went through.

First, the audio-recorded interview were transcribed. After transcribing the interviews, we attempted to extract information, meaningful to us and at the same time, understand the overall context. Figure 4.1 depicts the process of analysis, starting with interviewing and recording. It is important to notice that the process was not perfectly cyclic, but with each new interview new knowledge was added to our analysis, until we reached a saturation point where we could not learn anything new. Moreover, the categorisation was performed once every interview was collected.

Since we performed our interviews in two different languages, in 3 countries, we first performed a language reconciliation process. We did it because 4 of our interviews were entirely in French, but we needed to perform a deep analysis in English. To do this, the languages needed to be reconciled and this was accomplished by translating the interviews from French to English manually, with the goal of maintaining the original meaning of each phrase even if the exact wording had to be changed. Once we had a uniform set of transcripts, we needed to prepare the data for analysis.

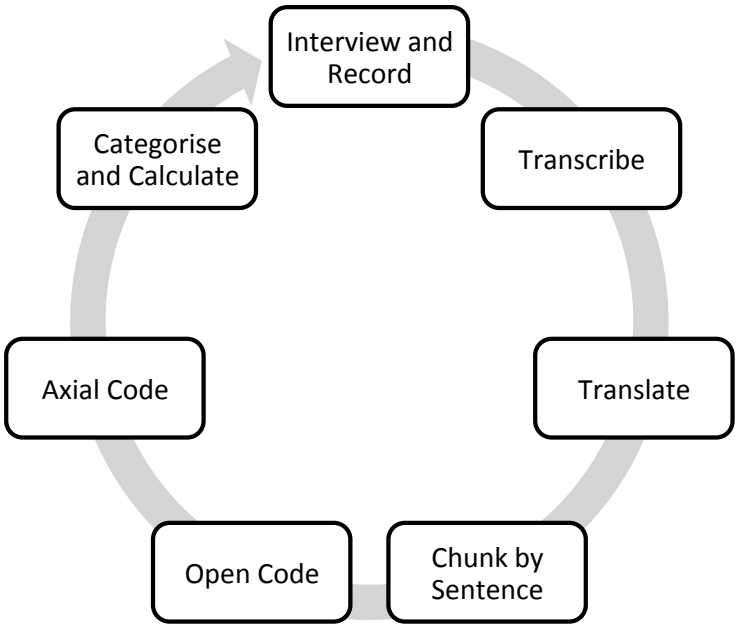

Figure 4.1 – The interview data collection and interview analysis process.

The first step of our analysis process consisted of performing a sentence-based open coding. In other words, we first split every transcript in sentences and then assign labels for each sentence. It is important to note that the chunks of data we chose were sentences and not fixed-size chunks of text, however, some sentences were split manually into a few smaller phrases, if we considered the sentence to carry multiple distinct meanings or topics. In order to accelerate the process of sentence splitting, we created a software tool that helped us along the way, saving some valuable time. The tool used a simple computational linguistics approach to split sentences by punctuation and was quite straightforward as it did not try to extract any meaning nor did it try to split into phrases. There were a total of 1947 sentences in our final, transcribed, interview dataset, with an average of 130 sentences per interview and 47077 transcribed words. The next phase of our analysis required performing so-called open coding. Open-coding is a process where a scientist reads through the data several times and assigns short labels for each data chunk. In our case, for each sentence we applied a 1-3 word label. The open coding process requires the label not to be merely a description, but be a higher-level one and attempt to describe the meaning of the data chunk, for example, by answering the question *"what did the interviewee actually mean when he said that?"* Figure 4.2 shows a part of our Excel file used for analysis and open coding procedure.

Interview	Phrase	Open Code	Category	Code	Sub-Category	Descriptive Category
10	How can I balance the load between the main lines of the execution?	Load Balancing	Orchestrate	Load Balancing	Method	Orchestrate - Method
10	And for us to understand what the problem in the load between is, the load balancing between the many lines of execution, we need to know which is the line of execution that's being kept behind.	Load Balancing	Orchestrate	Load Balancing	Method	Orchestrate - Method
10	That's why I was suggesting you highlight a color, you know, so that, if you have a very strong red close to the synchronization point, and some things up there, you have to be able to stop the execution in that point, and query what is the issue with those threads up there.	Visualize Load Balance	Techniques	Visualizing	Understand	Techniques - Understand
10	You know?	Visualize Load Balance	Techniques	Visualizing	Understand	Techniques - Understand
10	Maybe not one.	Visualize Load Balance	Techniques	Visualizing	Understand	Techniques - Understand
10	You know, maybe you cannot query one in specific, but you have to be able to see these things, live, you know?	Visualizing a Running Program	Techniques	Visualizing	Understand	Techniques - Understand
10	How all the threads are moving in the program, especially if this program, you can see it live in source code.	Visualizing a Running Program	Techniques	Visualizing	Understand	Techniques - Understand
10	You know, you have to understand how things are being stalled.	Visualizing a Running Program	Techniques	Visualizing	Understand	Techniques - Understand
10	I myself am working on auto tuning.	Project Specifications	Environment	Project	Component	Environment - Component
10	So, I mean that has nothing to do with visualization for now, but I'm part of an FP7 European project called Autotune.	Project Specifications	Environment	Project	Component	Environment - Component
10	Automatic online tuning.	Project Specifications	Environment	Project	Component	Environment - Component
10	Let's say the technical objective of the project is to build up a tool that is not only a provider and a tracer, but it also tunes the code for you.	Project Specifications	Environment	Project	Component	Environment - Component
10	So, you would pass a piece of code to what we're calling the tool PTF for now.	Project Specifications	Environment	Project	Component	Environment - Component

Figure 4.2 – Open and Axial coding dataset, used during the procedure of applying labels and seeking relationships between codes.

After open-coding, we performed an axial coding where we attempted to relate the codes to each other, creating sub-categories that link to general themes (categories), as shown in Figure 4.2. In order for us to understand the initial themes, we performed a card-sorting exercise where we printed out all 582 open codes that emerged during the interviews on small, post-it size paper sheets each. We attempted to tangibly cluster them, reasoning and trying to understand why a particular categorisation made sense. We describe the categories in more detail in the following section.

We also performed a statistical analysis in order to understand the importance of each sub-category, as shown in Figure 4.3. We have extracted the number of occurrences of a particular sub-category within all the interviews, and whether the term is present in an interview or not. Then it allowed us to compute a percentage value of whether the interviewees mentioned a particular topic or not.

During the entire process of analysing and annotating we also collected a large amount of different memos. Memos were used to perform a deeper analysis than just a categorisation, we used them extensively in order to derive the implications for design and advise tool builders about the modern tools that support regular programmers in their endeavour to build parallel performance optimisation tools. Additionally, during the process of analysing the interviews we have also constructed some interactive web-based visualisations in order to aid us in the task of exploration and understanding of the interview transcripts.

In is important to note that the process of hermeneutic analysis of interview data presented above runs the risk of subjective interpretation; in order to reduce some of the bias and strengthen our results we have presented the results of the analysis by

4	People	Ease of Use	4	4	27%	Ease of Use	Human Factors
5	People	Beneficial for Programmers	34	8	53%	Utility	Human Factors
6	People	Collaboration	39	10	67%	Collaboration	Collaboration
7	People	Company Reputation	1	1	7%	Organization	Organization
8	People	Error is Human	3	2	13%	Human Error	Human Factors
9	People	Level of Expertise	15	6	40%	Expertise	Experience
10	People	Human Limits	1	1	7%	Human Limits	Human Factors
11	People	Education	12	7	47%	Education	Education
12	People	Limits of Visualization	9	2	13%	Visualization	Human Factors
13	People	Visual Cues	9	2	13%	Visualization	Human Factors
14	People	Locate POI	1	1	7%	Point of Interest	Behavior
15	People	User Experience	1	1	7%	User Experience	Human Factors
16	People	Indication of Intent	1	1	7%	Indication of Intent	Human Factors
17	People	Long Project	4	1	7%	Mental Fatigue	Human Factors
18	People	Fun in Development	1	1	7%	Entertainment	Human Factors
19	People	Mental Fatigue	4	1	7%	Mental Fatigue	Human Factors
20	People	Inexperience with Parallel Tools	15	6	40%	Expertise	Experience
21	People	Human Error	3	2	13%	Human Error	Human Factors
22	People	Lack of Cues	2	1	7%	Cues	Human Factors
23	People	Customer Feedback	7	1	7%	Customer Feedback	Collaboration
24	People	Framework Adoption	1	1	7%	Technological Adoption	Behavior
25	People	Taking Initiative	1	1	7%	Initiative	Human Factors
26	People	Learning on the Field	36	9	60%	Active Experimentation	Behavior
27	People	Professional Work	36	9	60%	Active Experimentation	Behavior

Figure 4.3 – A snippet of the summarised analysis, showing various statistics about each sub-category along with major category and an open code.

e-mail, to most of the interviewees, and in person, to some. The interviewees have generally agreed with the results of our study, presented in this chapter. Furthermore, some of the interviewees have expressed the interest in our research and have contributed to several experiments discussed in this dissertation.

4.3 THEMES, CATEGORIES AND CODES

This section includes the results of our interview analysis categorisation, describing major and sub-categories that have emerged during our analysis and careful series of card-sorting exercises. The eight distinct major categories are related to people, understanding, orchestration, environment, goals / problems, resources, techniques and tools. For each major category we also linked open codes by a sub-category, representing a semantic relationship between the major category and the code. Below we describe each sub-category.

To help the reader in understanding our analysis, we present the categorisation in a series of simple bar charts. Each bar illustrates a single open code and shows the proportion of interviewees that mentioned or extensively spoke about the topic during our interviews.

Figure 4.4 shows the drill down of the environment category and its sub-categories. It represents the environment related concepts in which developers try to accomplish

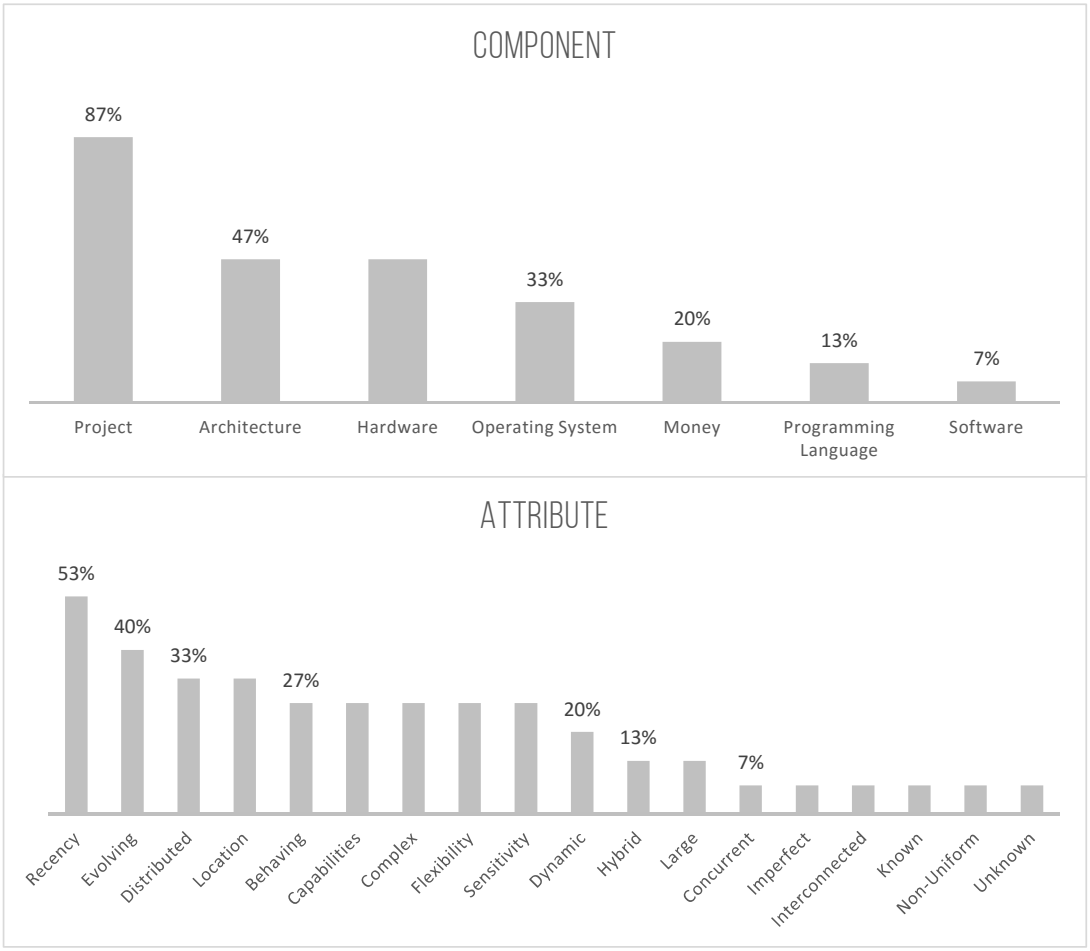

Figure 4.4 – Categories and Codes related to the **"environment"** theme.

their work, whether it's a company, a particular hardware configuration or a particular architecture.

The major category 'people' can be shown in the Figure 4.5. It represents people-related concepts such as human factors, collaboration, experience and education.

Figure 4.6 depicts the drill down of the major category about understanding. It represents a series of concepts explaining how people understand parallel programming and what kind of things they are trying to understand. Sub-categories include things such as hardware, bugs, components or interlinking.

In the Figure 4.7 the reader can see the drill down of the major category titled 'orchestration'. It represents a series of concepts explaining how people understand parallel programming and the kind of things they are trying to understand. Sub-categories include concepts such as hardware, bugs, components or interlinking.

As depicted in the Figure 4.8, the major category 'resources' is very specific. It

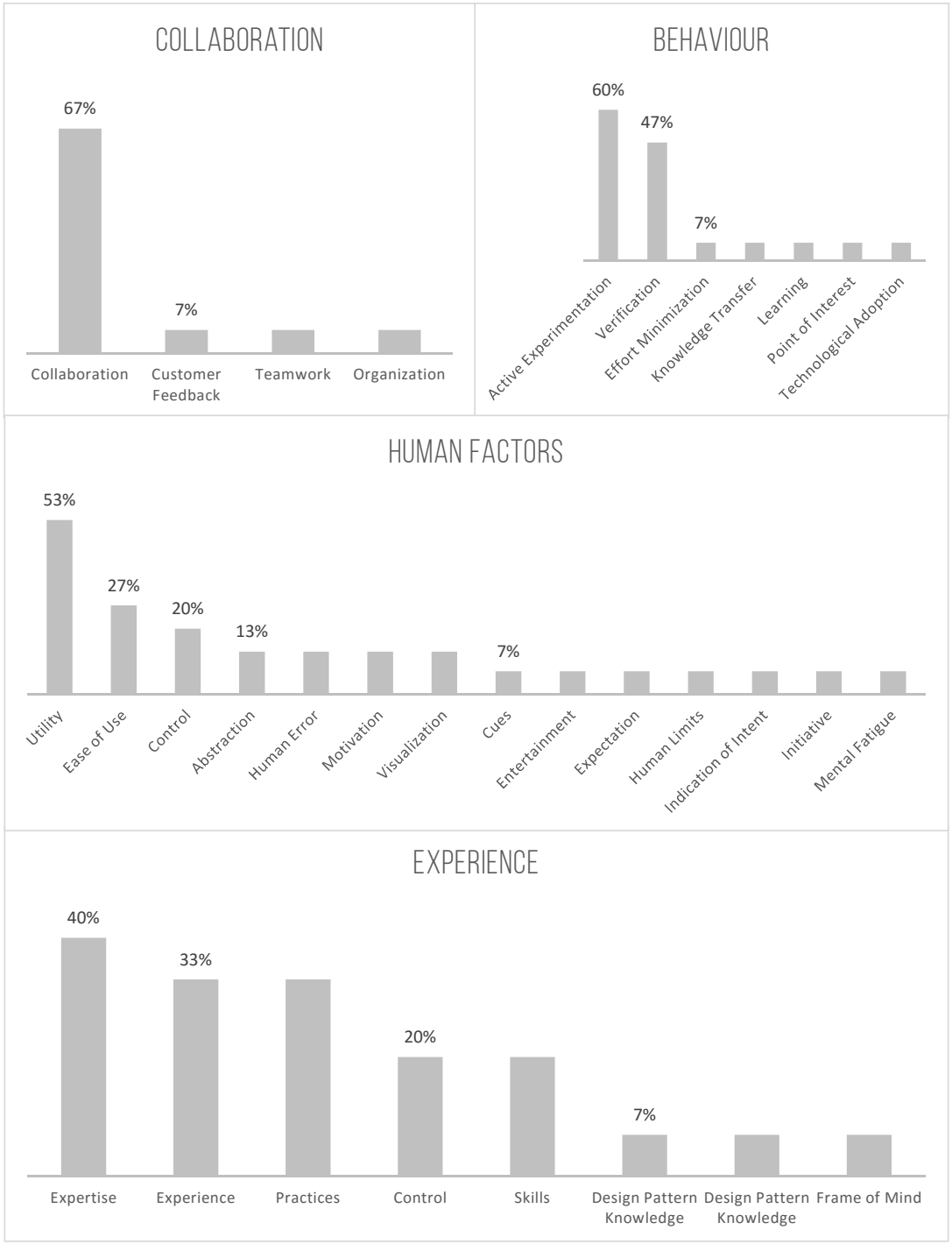

Figure 4.5 – Categories and Codes related to the **"people"** theme.

represents resources-related concepts, whether the resource is hardware, software and what types of properties each resource possesses.

In the Figure 4.9 the reader can see the drill down of the major category about goals (problems). It represents a series of concepts related to the types, contexts and

Figure 4.6 – Categories and Codes related to the **"understanding"** theme.

properties of various goals or problems that programmers seek to solve.

The Figure 4.10 shows various codes and sub-categories related to the major category of techniques. It represents the concepts related to the set of techniques developers use to solve the problems or achieve their goals.

Figure 4.7 – Categories and Codes related to the **"orchestrating"** theme.

Lastly, the Figure 4.11 depicts the drill down of the category about tools. It represents concepts related to the tools developers use in order to achieve their goals and the characteristics of these tools.

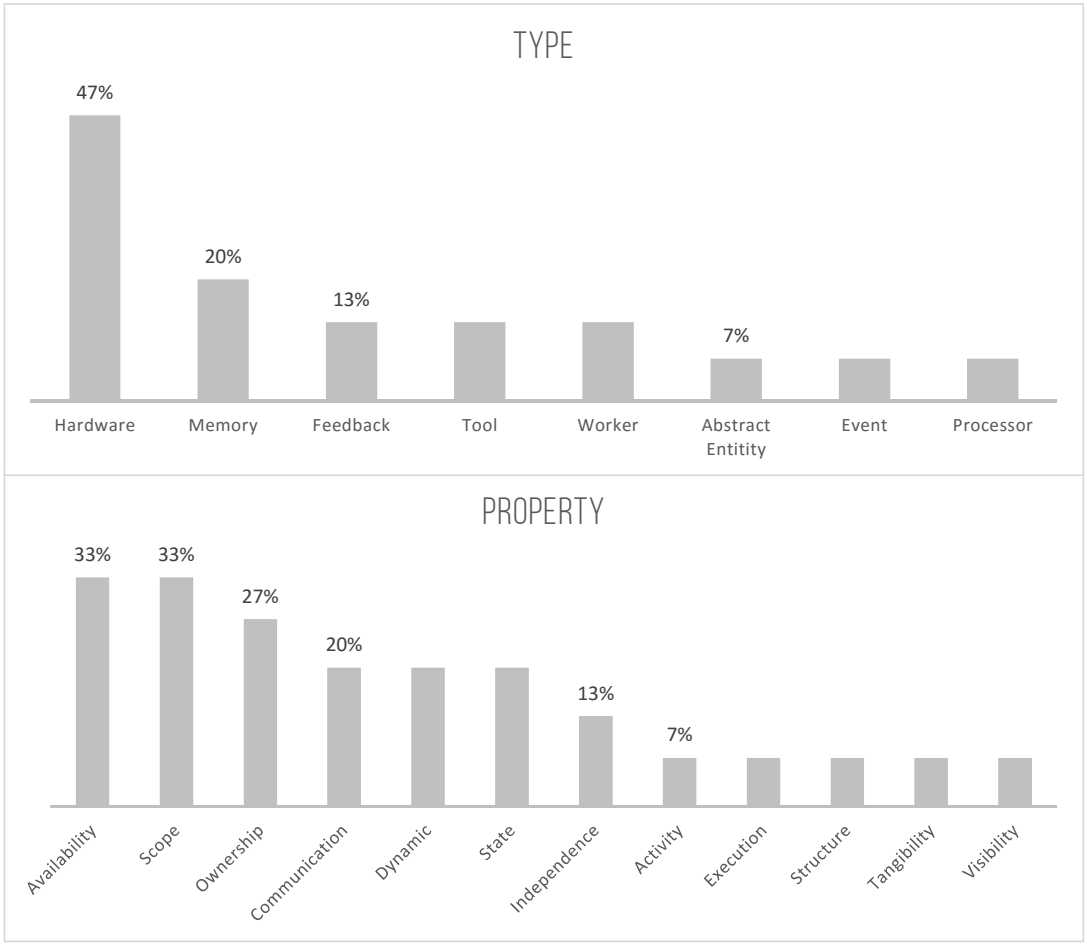

Figure 4.8 – Categories and Codes related to the **"resources"** theme.

Figure 4.9 – Categories and Codes related to the **"goal/problem"** theme.

Figure 4.10 – Categories and Codes related to the **"techniques"** theme.

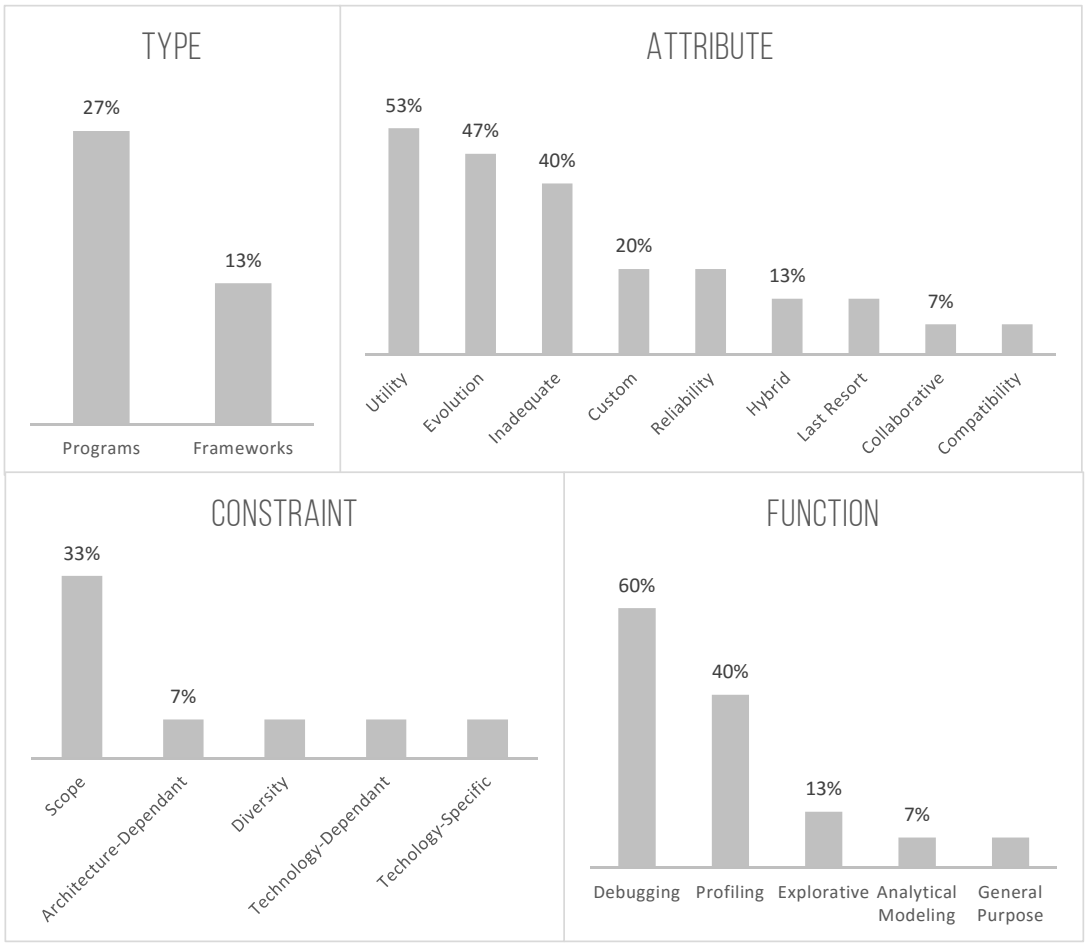

Figure 4.11 – Categories and Codes related to the **"tools"** theme.

4.4 DEVELOPING PARALLEL SOFTWARE

During our study we identified three major poles of activity in a developers work. Developers attempt to: 1) Meet their own goals and the goals of their organisation, 2) Understand what is going on with the workers (threads, processes, machines). Where are they, what are they doing, are they performing well? 3) Orchestrate the group of 'workers' and make them work efficiently together. In the following sub-sections we examine these three poles.

4.4.1 CONTEXT FOR DEVELOPMENT

The developers interviewed were mostly based in industry and every organisation had their own unique set of constraints and concerns. Their working environments (both physical and virtual) are composed of various components such as software specifications, budgeting and deadlines, human resources, hardware and software constraints, programming languages, software architecture, the actual code, etc. An interesting aspect of the study is that the programmers interviewed did not dwell on programming paradigms and talked about their experience across different programming languages. We know, however, that many were using imperative languages such C, C++, Fortran or Java, and some used hybrid languages such as C# or Matlab.

Throughout the interviews, developers often talked about complexity, and specifically about the complexity of the software: large systems tend to become larger and ever more complex as new features are introduced. One developer, commenting on a major web search engine stated:

> **Interviewee**: "*When you have a 20 million line app, no one can fit it in the brain.*"

The complexity of many of these systems is such that no one person understands every detail of the system. The knowledge is shared between people in the organisation, and distributed across various information systems.

While developers often worked with extensive tool support in the form of IDE's, this support does not extend specifically to debugging and optimising parallel software. As a result, developers tend to rely on intuition and resort to 'thinking really

hard' about the problem. There are two aspects to reasoning about problems; on a conceptual level developers want to focus on the problematic elements of their program and abstract away elements not immediately relevant to the matter in hand. On a concrete level, there are also issues of experimental noise; they may wish to replicate the problem in the simplest possible form in order to clearly identify the source of the issue and test possible solutions.

> **Interviewee**: "*(...) sort of a test bench, where we just took that problem piece of code and then just on that algorithm, just enough of the code to be able to start that algorithm and run it or just enough of the code to run it to get the results and to be able to verify that they were accurate.*"

However, developers are often confronted with various and sometimes unexpected problems. These problems can occur on different levels: in a particular function in their code, in an external library, on a particular machine or a cluster, or even on an abstract architectural level.

Most developers who develop parallel software do so in order to obtain better performance. Therefore, performance becomes a design requirement. In consequence, developers do not think of parallel performance optimisation as a separate process, but as a correctness issue, a particular type of bug. This contrasts with traditional software development where the two concerns are generally more easily separated.

> **Interviewee**: "*In the normal software development they usually forget about optimisation at the beginning and then look to optimise later. In HPC it is totally different approach: you design your code to be optimal.*"

4.4.2 UNDERSTANDING

In almost every interview, developers described methods they use which help them improve their understanding of the system they are working on, the environment or various problems that occur. There are numerous things they try to understand. For example, they may want to know which component is at fault (code path, thread, task), when and what caused the failure to occur. Alternatively, they may be interested in understanding the architecture of the software they are working on (patterns of

communication, design patterns, dependancies, hardware).

In order to improve their understanding, developers use a range of techniques. Some of these techniques are only possible with very specific tools whereas in other cases they may simply ask other people.

Table 4.2 provides a non-exhaustive list of techniques that were mentioned or discussed to some extent during the interviews.

% of Participants	Technique
60%	Active Experimentation
47%	Verification
40%	Visualizing
33%	Tracing
27%	Abstracting the Complexity
20%	Benchmarking
20%	Chain of Events Analysis
13%	Deadlock Detection
13%	Documentation

Table 4.2 – A non-exhaustive set of techniques for understanding and the percentage of interviewees who were talking about the subject.

In addition to this, there are various obstacles that developers have to overcome in order to gain a better understanding. Obstacles mentioned include the overall complexity, non-determinism, incomplete information, incorrect assumptions and lack of documentation.

A developer in a large organisation commented:

> **Interviewee**: "*Some classes were written decades ago, then somebody puts it in another product, and then to another product. And suddenly someone says that we're going to use multiple threads to do that, and all of the sudden this legacy code has been dragged into the 21st century.*"

In this case, the programmer used an external library (a piece of code) written by someone else some decades ago. The new developer made an erroneous assumption about thread-safety as relevant information was not available to him, resulting in a poorly performing and buggy program.

INADEQUATE TOOLS

The majority of developers we interviewed expressed dissatisfaction with available tools. Three developers we interviewed clearly stated that the tools were very poor a few years ago, but gradually getting better over time. Five of the interviewees asserted that they do not use any performance analysis tools at all.

> **Interviewee**: " *(...) Take the lock again, commit everything back. But the problem is, at least in C++ and C, there's no great tools or mechanisms to support that type of thing, or to even find out that you have those problems.*"

In some cases, this discontent even results in a person, in the case of the next quote, a computer-science expert, not using tools and relying solely on intuition for the whole debugging and optimisation process.

> **Interviewee**: "*I think the only way I can do most of this debugging is stare at it and debug it in my head, which is the least effective way you can ever debug.*"

There seems to be a gap between the results that performance analysis tools provide and the information that developers seek. Developers are usually interested in a specific issue that they wish to diagnose. They see a symptom of a performance issue and they attempt to figure out what caused it. This is a challenging, and often time consuming process. This process of understanding is still poorly understood and developers use various techniques. For example, they might experiment, putting traces in code, recompiling over and over again, they might try to go back to the whiteboard and create some hypotheses of which component is failing, or use tools that assist them. The whole process is usually a combination of all those things.

> **Interviewee**: "*But I spent, I do not know, maybe a week just staring at the thing, watching it god knows how many times.*"

THE PROBE EFFECT

One of the most problematic barriers for understanding the performance of complex software systems, is the presence of the probe effect. The probe effect denotes an

unintended change to the behaviour of a system caused by observing (measuring) that system [53, 95]. In order to measure something in the program, an automated tool or a person adds additional code to record features of interest. This approach is called program instrumentation.

> **Interviewee**: "*And then, there's a question of how to observe the parallelism. [...] An observation brings problems which wouldn't be there if we wouldn't observe.*"

In order for a performance measurement system to be safe to use on production systems, it is generally considered that the performance analysis infrastructure should have zero probe effect when disabled and should not accidentally induce any errors when enabled [26]. However, most instrumentation techniques have a probe effect when enabled making it difficult to apply continuous monitoring of performance on production systems.

Developers are aware of the presence of potential probe effects and 27% highlighted the probe effect as an important issue, resulting in them not using any performance analysis tools on production systems. By slowing down the whole program, the behaviour of the program is altered and makes it difficult or impossible to track some performance issues such as race conditions. This is especially problematic when developers have to diagnose performance issues, as the instrumentation may slow down the whole program to the point where the performance problem is no longer observable.

> **Interviewee**: "*We had a number of interesting issues. The actual turning on the log it slowed everything down such as the problem would not be there with logging on.*"

INFORMATION REPRESENTATION AND RESOURCES

In order to cope with the complexity of software development, developers create abstract representations of the environment and contextualise the work they are performing. For example, interviewees talk about a global scope or a local scope. The scope represents a context in which they perform their tasks, allowing them to focus and reason on a particular level:

> **Interviewee**: "*So you have parallel programming at the GPU level,* [1] *at the CPU level at cluster level, and then, you know, maybe deeper even.*"

The global scope represents a 'big picture' of the system. For example, hardware, systems and software architectures. The global scope thinking allows developers to think about the problem in a more abstract way. The local scope represents a 'detailed picture' of a sub-system of the system the developers are working on. For instance, a developer might consider using a specific algorithm in order to optimise some part of a program.

As mentioned above, complexity can be overwhelming. Developers therefore use tools to offload knowledge into the world, a common strategy for people faced with complexity [123]. For example, several developers reported using UML[2] or sketching tools, even pen and paper, in order to externalise their knowledge by 'charting' the environment in this way:

> **Interviewee**: "*The only tools I used [during the project] were UML analysis tools.*"

However, while developers are focusing on a local scope, they may lack contextual information. For example, while with a modern integrated development environment (IDE), the developer may know what parameters are taken by a function they call, they usually do not know whether the function is thread-safe, what side-effects it produces or how well it performs. This is especially problematic in the case of functions or classes that are poorly documented. Developers stated that they rely heavily on documentation, augmenting available resources with internal wikis, and tend to use third party libraries that they know are well tested and perform well.

> **Interviewee**: "*I'll often look to use the Intel MKL*[3] *libraries, wherever possible. [...] But the documentation for it is really excellent.* "

Developers seek information relevant for the task at hand. For example, if a programmer designs a new feature which complements a bigger system, they need in-

[1] Graphics Procesing Unit (GPU) can be used by developers to perform some computation, usually number-crunching.

[2] UML stands for Unified Modelling Language, a standardised general-purpose modelling language in the field of software engineering.

[3] MKL stands for Math Kernel Library, a set of routines which includes highly vectorised and threaded Linear Algebra, Fast Fourier Transforms (FFT), Vector Math and Statistics functions.

formation about the existing elements of the system in order to achieve their goal. If the programmer develops a particular algorithm destined to work on particular hardware, they might need hardware specifications in order to create an optimal algorithm. The programmer needs information relevant to their task, but information is often fragmented and distributed across various sources (documentation, internal wiki, people).

4.4.3 ORCHESTRATION

The process of parallel programming is essentially a process of coordinating the efforts of different workers and balancing workload, which we can paraphrase as orchestration. The programmer attempts to distribute the workload across multiple worker nodes, to achieve optimal resource utilisation, maximise throughput, minimise response time, and avoid overhead. Programmers in different contexts will have different constraints and therefore might perform this orchestration differently. For example, a developer working alone in a small company, would want to reduce his costs in writing the software by reducing the time required to do so. On the other hand, a PhD researcher might concentrate more on the performance of their program, sacrificing time. This leads the developers to use different orchestration techniques, depending on the situation.

ORCHESTRATION MODELS

As mentioned above, programmers go through a process of orchestration, trying to find a solution that allows them to efficiently distribute the workload across multiple workers. In programming, as with many other crafts, reusable solutions for commonly occurring problems are often formalised. Some such solutions discussed during the interviews are also known, more formally, as design patterns. An advantage of such formalised design patterns is that they provide both a common conceptual model and a vocabulary for everyone who uses them, allowing programmers to reason about and solve problems more easily.

Interviewee: *"The goal was to decouple an image processing part as much*

as possible in a pipeline and partition the tasks on the different resources [machines].”

During the interviews, we asked people to talk about various challenges they encountered during their projects. They all gave at least one, high level explanation of the solution they implemented. The projects the programmers worked on were varied: from a small multiplayer game running on a single computer, to one of the biggest search engines in the world, running on thousands of computers. They talked about the way their program is orchestrated, about the architecture and about the high-level design.

Across these various projects and solutions that were put in place, similar orchestration patterns were applied. Surprisingly only some of the developers were familiar with formalised design patterns and named them. However, many developers we interviewed were essentially talking about a design pattern. Table 4.3 shows the percentage of developers we interviewed who were using knowingly or unknowingly a recognisable design pattern in their work.

Occurrences	Orchestration Model
40%	Producer / Consumer
33%	Active Object
27%	Scatter / Gather
20%	Pipes and Filters
20%	Futures and Promises

Table 4.3 – A non-exhaustive set of orchestration models and the percentage of interviewees who were talking about the subject.

For example, one of most commonly mentioned design patterns is the Producer/Consumer, a design pattern that is used to decouple workers that produce and consume data at different rates. Such decoupling grants the programmer flexibility in how they partition the workload in a scalable manner. Another example of a widely used design pattern is the Active Object model. This pattern allows independent threads of execution to interleave their concurrent access to data modelled as a single object. This model allows the developers to simplify synchronisation complexity and transparently leverage available parallelism.

While design patterns are in part formally defined in the literature, in the context of parallel programming, their significance is in the way that they provide **orchestra-**

tion models. An easy metaphor to understand this, is to consider that a developer who writes parallel programs does not write recipes (as often taught to first year computer science students), but manages a project with several workers working for him. The developer attempts to orchestrate these workers in an optimal way, considering the available resources and tasks at hand. In the parallel programming context, such workers can be threads, processes, machines, domain-specific classes, etc.

> **Interviewee**: "*It's like seeing a team working. Sometimes the team works better when you have some people that work worse for some reason.*"

Orchestration models are a type of design pattern employed in parallel programming, and they span different contexts (or scopes) on which developers are focusing. In software engineering design patterns are often related to reusable code skeletons for quick and reliable development of parallel applications [149]. In contrast, orchestration models are more abstract, spanning not only patterns related to software architecture, but hardware and system architectures as well.

Programmers tended to refer to specific orchestration models when describing the way they think about the design and implementation of concurrent systems.

DIFFICULTIES WITH ORCHESTRATION

While the orchestration models provide an initial basis for design, helping the programmers to reason about and implement their system, there are many issues associated with the process of orchestration. In this section we highlight some of the difficulties described by the developers. To illustrate our point, we start by giving a couple of examples of problems the programmers encountered.

> **Problem:** Novice programmers (computer science students) were implementing a concurrent game, running on a single multi-core machine. They ran into critical deadlock and slowdown issues due to lock contentions.
>
> **Adopted Solution:** Students were forced to reengineer the whole game and completely understand the interactions between components. By removing redundant locks and following a critical section pattern for ev-

ery component, they achieved a correct and fast implementation. They drew a schema of interaction of various components, mainly in order to understand the activity of workers at any given moment, and the resource usage and sharing.

Problem: Expert developers were implementing a system, allowing them to perform computationally intensive tasks on a massively parallel GPU. They ran into a series of subtle problems, the most common being branch misprediction. Branch misprediction occurs when a processing unit (CPU or GPU) mispredicts the next instruction to execute, which impacts adversely on overall execution time.
Adopted Solution: They ran tests to identify which branches were taken most often and adapted the code accordingly.

Problem: Expert developers were implementing an equivalent of an OpenMP "parallel for" loop. They were building a library for other developers to use and ran into a problem where their implementation did not perform as well as expected in production, far below the theoretical performance. They had been very focused on the implementation and theoretical performance and did not consider the context, in this case, the influence of the operating system which negatively impacted on the performance.
Adopted Solution: They adopted a clever solution, where workers would steal work from other workers (work-stealing). This allowed them to have an implementation which was more dynamic in nature and could cope with the uncertainty introduced by the operating system.

In the examples above, as in many other instances observed during our interviews, developers implemented the system, but noticed a slower than expected performance or a critical bug during the verification phase. They encountered undesirable side effects of both a deterministic and non-deterministic nature.

> **Interviewee**: "*Because it just works most of the time, and you occasionally just see some corruption*"

Non-deterministic issues may manifest themselves in various forms, and, most importantly be unpredictable and difficult to reproduce. The programmer's job is to prune as much nondeterminism as possible [96]. Moreover, in many cases, developers had to go back to the drawing board and reengineer, at least partially, their system.

"OPTIMAL" SOLUTIONS

Developers want to reach a working solution which is close to the best achievable result given their organisational constraints. Essentially, they are trying to find an orchestration model that allows them to build a system with satisfactory performance. The "optimal solution" in this context represents an ideal, a scalable and practically feasible solution that programmers strive to achieve (rather than theoretically optimal).

> **Interviewee**: "*There is not an analysis tool in existence that helps to ensure that we're getting to an optimal solution.*"

Picking a satisficing orchestration model is not an easy task as it depends on various constraints. We have previously illustrated some of the constraints, including the correctness or performance, but an orchestration model, as opposed to a design pattern, can also have more general constraints, such as usability. To illustrate this, consider the following quote from an expert developer who worked on a "parallel for" loop abstraction, from the third example.

> **Interviewee**: "*Next thing we worked on is a parallel for. A parallel for by itself it's not terribly hard [to implement], but there is different types of workloads... So, analysing every workload and seeing how a parallel for would perform and what are the different jobs that a user could actually use. [It] turned out to be a herculean task, because most people use it for a very simple thing. For example: I have two arrays of numbers and I use parallel for to sum them. So now, there's work which is too small in every iteration, which means we can not introduce any overhead. But the same parallel for could be used to do much more complex problems: tons of work within every iteration. And, even worse, they could use*

it for raytracing, for which in every iteration the work is different."

Essentially, the developer explained to us that in addition of having correct and fast implementation, their system had to be used in many different ways by developers, and thus being usable and flexible. Once again, they had to go back and reengineer a part of the system to cope with this requirement.

SUMMARY

Whether explicitly or implicitly, developers are thinking about their programs in terms of orchestration models. The majority (60%) of developers we interviewed talked about load balancing and optimality. They want to orchestrate their programs in a efficient manner, coordinate workers and distribute resources. There is a clear need for tools that help developers to go 'back to the drawing board' and analyse the underlying architecture, the orchestration model, of a program upon encountering performance problems.

4.5 DISCUSSION

In this section, we consider and discuss potential implications of the study regarding parallel programming practices and the design of tools to support these.

Cover both correctness and performance. While our study initially intended to focus only on performance improvement, during the interviews it quickly became clear that that performance concerns cannot easily be separated from program correctness, as is the case in traditional programming tasks. This is due to the nature of the problem itself, as programmers write parallel code in order to leverage the resources and increase performance. Good performance thus becomes a design requirement.

This observation leads us to consider a range of implications. Firstly, tools that support parallel programming activity should consider both correctness and performance issues at the same time. This is different from traditional programming where we can see two major types of tools: profilers which are used for measuring and analysing the performance of a system, and debuggers used for diagnosing correctness problems. Secondly, this observation also means that it might be possible to apply research

on debugging in software engineering to parallel performance problems, such as the information foraging perspective [94, 133].

Support active experimentation and tracing. Developers are not happy with currently available tools for parallel programming. At least one interviewed developer in five was not using any performance analysis tools at all and attempting to reason about the performance using tracing and active experimentation.

Therefore we suggest features supporting incremental running and testing of programs should be included in software performance analytics systems. It has been previously noted that such features help developers to be more productive during the debugging process [60, 58, 131]. Hence, enhancing software performance analytics systems with immediate feedback and exploratory search could result in more productive performance analysis. As an example, consider a visualisation system of performance data that supports an exploratory search process, allowing developers to localise performance bottlenecks and hence narrow the range of possible causes. Moreover, such a visual analytics tool could aid program comprehension and help reduce bugs [56]. While this strategy is applicable to more traditional programming tasks, parallel programming introduces more parameters which might be varied, such as conducting multiple runs with different numbers of threads, or changing processor affinity. There appears to be a gap between what developers need and what current tools provide. Most of the current tools are designed and built by computer scientists in a bottom-up, data-driven fashion. Rather than taking bottom-up approach, building tools and sticking a nice barchart on top of the captured performance data, we must build meaningful tools what are tailored to developer's actual needs. Ideally, performance tools should not only be useful, but also easy to learn, require a minimal effort from developer, assist them in experimentation, encompass contextual information and highlight points of interest.

Consider the environment and the context. Developers deal with very complex environments and create abstractions and contextualise the work they are doing. However, they seem to lack support for integration of various sources of information, even simple things such as a link to the page of the company's wiki explaining the architecture of a particular library or server configuration.

We suggest that developers of software performance analytics systems keep in

mind that developers work in a highly interconnected system. The interviewees expressed concern about lack of context, bad documentation or legacy code. This can be solved by integrating contextualised knowledge into tools. For example, if a developer uses a particular library, it is useful for him to be able to have easy access to documentation, have it automatically checked for thread-safety, or present some information about performance of the piece of code they are about to include.

Consider the probe effect. Developers are concerned about the probe effect. Most tools rely on code instrumentation, and hence alter the behaviour of the program. In some cases, this leads to developers avoiding using any tools and relying solely on intuition. The probe effect also makes developers avoid using performance analysis tools for continuous monitoring of production systems. Once the system is deployed, developers desire maximum performance of the system and anything that might slow down or impair a deployed program is avoided. This makes it difficult to measure the performance, capture and diagnose performance issues on production servers, when it occurs. It is especially problematic with non-deterministic issues which may occur only rarely. For example, the Ptolemy II Project developed a process that included a code maturity system, design reviews, code reviews, nightly builds, regression tests, and automated code coverage metrics. They also wrote regression tests that achieved 100 percent code coverage. However, after four years of use, a deadlock was encountered [96].

We suggest that the probe effect issue must be considered from the beginning while building performance analysis tools. It may be possible to reduce or avoid the probe effect by using a combination of hardware counters and operating system events. [176]. While such approaches are less relevant from a HCI perspective, an important challenge is how to make this data meaningful to the programmer; it must be possible to relate the low-level data back to the structure of the program.

Make use of orchestration models. During the interviews, developers discussed a range of design patterns and architectures they used throughout their programs. An interesting design direction would hence be to leverage orchestration models in the design of software performance analysis tools.

Developers seek to pick a suitable orchestration model for a particular scope. It can be a design pattern, a way to split an array in a function for a parallel loop, or even

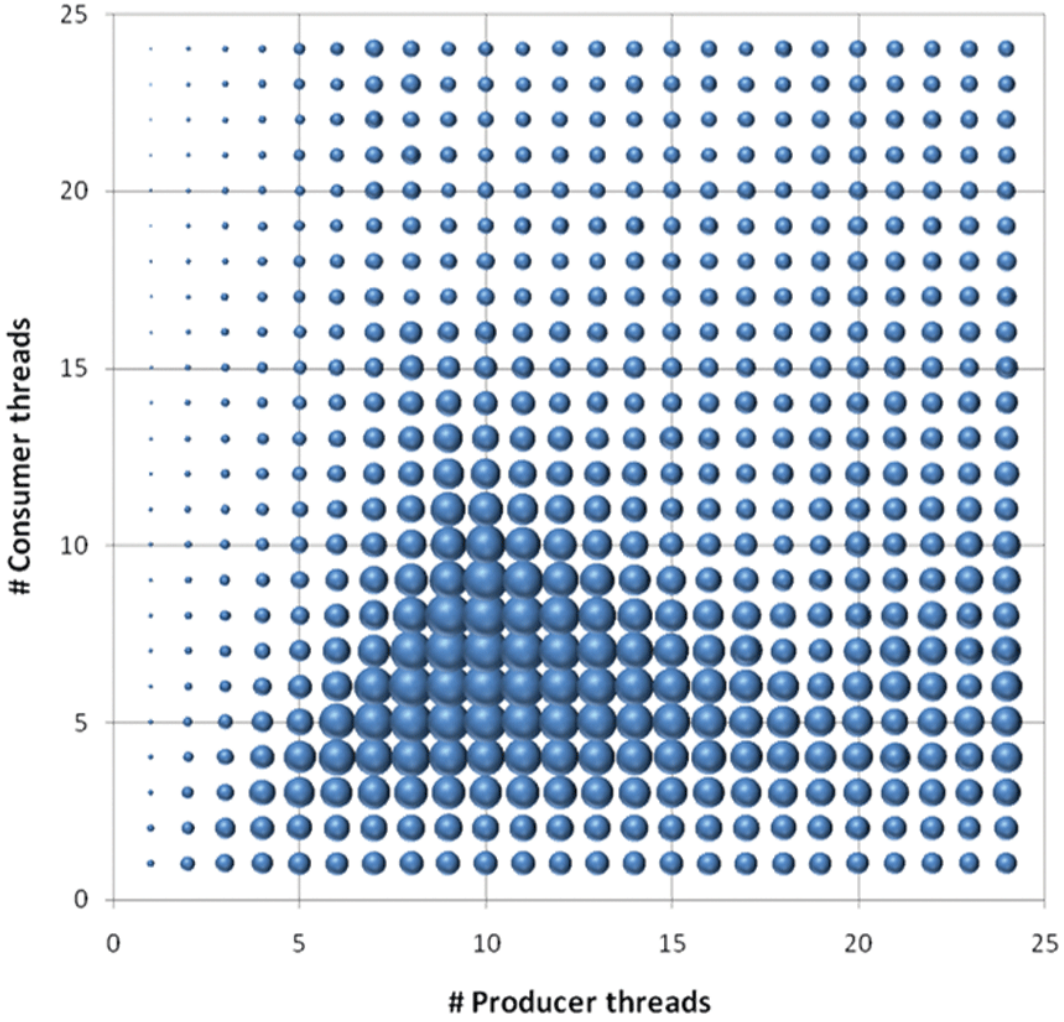

Figure 4.12 – Sample graph for measuring scalability of the same algorithm for different workloads [72].

a cluster configuration. A good tool should help them in this endeavour of "thinking parallel", which is also the number one point on a recent list of challenges in parallel computing [106].

One can think of numerous ways to take advantage of orchestration models. One direction would be to design visualisations which are specific to particular design patterns, and which emphasise features of interest within these, illustrating the performance data with respect to the pattern. One example of such a visualisation can be found in Sutter [72] and depicted in the 4.12, in which scalability of a given algorithm is charted for a range of different numbers of producer threads and consumer threads, within a producer-consumer orchestration model.

Help in finding an optimal solution. As many developers seek to get as much performance as they can, providing that in doing so the code remains maintainable and reusable, we need to help them in this task.

Ideally, this can be done via automatic parallelisation, which is the Holy Grail of parallel computing and aims to automatically parallelise the work, thus not requiring the developer to know about parallel programming at all. However, since it does not seem to be realistically possible in the near future, there are many other things that could be done, for example, predicting the performance of an application [79, 174].

Provide support for scope exploration. We have noted that developers consider the notion of scope of their orchestration. Additionally, when faced with a challenge, developers must often go 'back to drawing board' and reconsider their orchestration model. This process requires them to look at the problem from different angles, essentially being able to consider orchestration models at various levels of abstraction. By providing the ability to developers to interactively change the scope (core, CPU, thread, and upwards) and re-sample measurement data accordingly, developers would have additional flexibility in the challenge of diagnosis and re-orchestration.

4.6 CONCLUDING REMARKS

Parallel programming is an important, but very complex activity. In this chapter we have identified and explored some of the challenges involved in parallel programming, in particular the challenge of parallel performance analysis.

As well as illustrating the way in which correctness and performance issues are interwoven in parallel programming, the way in which the probe effect influences their performance analysis and debugging behaviour, and the way issues surrounding the complexity of the task and environment, the study has looked at the role of orchestration models in parallel software development. While some of the issues identified are apply to more traditional software development, in all cases additional dimensions are introduced by parallel programming.

Developers of parallel software have to meet their goals within a complex environment. In some sense, they are similar to factory managers and have to orchestrate multiple workers and manage them as a team. This metaphor of parallel program-

ming is different from traditional programming, where developers write a recipe for a unique worker, here, they have to deal with often unpredictable interactions.

The study can help to inform the discussion on potential tool support for developers, and particularly the design of performance analysis tools.

Chapter 5 explores the problem space in systematic fashion; identify the major types of problems that programmers might encounter, and the information they might exploit in their diagnosis.

In Chapter 6 and 7 we explore how we might build on such a model by focusing on the specific example of data locality. The design, implementation and evaluation of a performance visualisation for data locality is presented, building on insights and requirements emerging from the study.

CHAPTER 5 MODELLING THE DIAGNOSIS

In our effort to establish a framework to help parallel performance tool developers in dealing with performance problem analysis, we need to create an effective model which can be used by practitioners in their endeavour. In order to effectively help diagnose parallel performance problems, we need to know what those problems are; while we have identified some of the problems in the previous chapter, it is necessary to perform a more comprehensive analysis of the field literature.

At the point of writing, no formal and well accepted taxonomy of parallel performance problems exists so far, that we are aware of. Hence, we present an initial taxonomy of parallel performance problems. These are classified into seven interrelated categories, six of which we consider in detail in this thesis[1]. Each category contains several specific performance problems that can arise in parallel programs.

There are many possible ways to conceptualise parallel performance problems, and several of the categories in the taxonomy can be seen as overlapping to some extent, or to interfere with one another; a given issue with the software might also lead to multiple performance problems. Furthermore, certain types of parallel performance problems, such as those arising from the design of the software architecture, are not easily characterised. There is a strong bias in our model towards problems that lend themselves to measurements of the executing program.

In terms of measurement, hardware counters such as cache misses and memory controller bandwidth, together with operating system events such as context switches and I/O writes provide raw data which can be processed to produce performance metrics. During the process of identifying performance problems, a developer can interpret these performance metrics and relate these to the performance of various

[1] One of the categories (I/O) is conceptually distinct and not covered in detail in this thesis, for reasons which are discussed later.

components of their own mental representation of the program.

Building on the taxonomy, we present an observational model which identifies a set of potential symptoms which may be associated with particular problems within the taxonomy. Additionally, a series of discriminating (contrary) symptoms are also proposed. Two accompanying studies are used to validate and explore the model. We investigate the relationship between: (a) performance data we can collect, measure or calculate such as operating system events or performance counters and, (b) the process of identification of performance problems. In other words, to what extent can programmers of multi-core systems effectively identify specific problems given a set of measurable features?

The aim is not to discover the "one true model" but rather a useful model with reasonable coverage of the problem space which can be used as the basis for discussions around tool support. To this end, we have cross-validated the model in a study with 10 experts, constructed as an inter-rater annotation exercise [166]. The experts annotated various observations as being indications, contraindications or irrelevant to the performance problem presented. While F.D. Roosevelt quipped that *"there are as many opinions as there are experts"*, the expert study helps to identify both those areas with high agreement, improving confidence in the objective validity of those parts of the model, as well as those where there is disagreement. Disagreement may indicate the need for further research, but might also arise in cases where context plays a strong role in the significance of a particular observation, requiring a more detailed breakdown of the observations and their meaning.

We accompany the expert validation exercise with a survey using this taxonomy to assess the familiarity and frequency of these problems among developers, to further help in identifying good candidates for improved tool support.

5.1 PROBLEM TAXONOMY

A taxonomy of parallel performance problems provides us with a common foundation and vocabulary for discussing the diagnosis of parallel performance problems and provides concepts which can help in the design of tools to support their detection. In this section, we present one possible taxonomy, developed iteratively and derived

from several sources. The initial starting point was an earlier qualitative study [12] in which interviews with parallel software developers were fully transcribed and coded following a qualitative research methodology, resulting in 582 open codes. While the focus of the qualitative analysis was a broad characterization of needs and practices, as a by-product of the analysis, a range of different types of performance problem emerged, including load balancing, lock contention and saturating memory bandwidth. In parallel, we examined two main classes of literature on software performance optimization for multicore. We examined both practical materials and manuals aimed at software developers [4, 2, 1, 8] and books and papers from the research literature on performance of parallel programs [7, 24, 158, 179]. These sources were used to provide material for both the taxonomy of problems, and the model relating observable data to these.

A useful source of information on multicore performance problems is the Intel software optimization manual [4]. It specifies five areas for optimization of parallel programs: thread synchronization, bus utilization, memory optimization, front-end optimization and management of shared execution resources [4]. Specific performance problems listed include false sharing, spinlocks, sharing updated data between cores, and limited memory bandwidth. Elements from the literature informed our descriptions of task granularity [31], lock contention[16], low-work to synchronization ratio [142, 73], data sharing [4, 71], load balancing [9], TLB locality [57], DRAM memory pages [41], NUMA [4], false sharing [8], and shared resource contention [179].

We have worked to make the emerging taxonomy and model more coherent and bring it into line with the terms used within the computer architecture and parallel programming literature, and relating each to potential observations. Short descriptions of each problem were also produced in this way, with reference to the literature. Further review was done sequentially with two domain experts (in building parallel software and solving performance problems in parallel software). Each expert reviewed both the taxonomy and observations, and asked to consider which were relevant and useful and which are not, as well as identify any missing items. This was done in order to validate and stabilize the taxonomy and model before progressing with the studies described within the paper. The final output of this process was the taxonomy, the short descriptions of each problem, and 110 observations relating to

these[2].

A complication is that most performance problems that can exist on a single core can also exist with multiple cores. Rather than trying to describe all the single-core and multicore performance problems together, we instead focus on the problems that arise from the interaction of multiple cores. It is difficult to draw clean lines between related types of performance problem, and some parts of our taxonomy might overlap in places. However, in the development of the taxonomy we have sought to provide coverage of the most important parallel performance problems in shared-memory multicore systems. Another issue is that it is not always clear where in the taxonomy to place particular problems. For example, exceeding memory bandwidth could be a resource sharing problem or a locality problem.

It is important to note that when we refer to various performance problems, we are referring to features of the parallel program that limit performance. For example, in some applications exceeding available memory bandwidth may be an innate feature of the application area and algorithms used. Many sparse linear algebra problems operate on huge sparse matrices and are inherently limited by memory bandwidth. Using every bit of available memory bandwidth may actually be a partial solution to dealing with these huge data structures. When we refer to the resource limitations as being a performance problem, we more precisely mean that they limit performance. It is difficult to improve performance without addressing the problem. Similarly, we regard large amounts of sequential execution as a performance problem, but in some cases this may be inherent in the application. Sequential execution is a performance problem in the sense that it limits parallel speedup. But there is no guarantee that the problem can be solved in any specific program.

5.1.1 SCOPE OF TAXONOMY

Multicore computing encompasses a wide range of architectures with many different features. For example, multicore digital signal processing processors often have low-level hardware features such as software-managed on-chip memories. Multicore

[2]The coding from the preceding qualitative study, experimental materials (category and problem descriptions), and notes regarding derivation of the taxonomy are available from `http://www.scss.tcd.ie/ManyCore`

network processors have special features to accelerate packet processing. Some people regard graphics processing units (GPUs) to be multicore processors.

For the work presented here, we restrict the scope to perhaps the most ubiquitous class of multicore architectures: A machine with multiple cores, a shared memory with hardware cache coherency, and which runs an operating system. The main multicore processor on almost all server, desktop and laptop machines belongs to this class [71]. Many embedded multicore processors, especially those from ARM, also follow this model.

We assume a shared-memory programming model based on threads, with locks, synchronization barriers, semaphores, critical sections and similar mechanisms as the main synchronization mechanism[3]. The machine might have two or more CPU chips, each containing multiple cores. But all cores operate on a single shared consistent memory (where consistency and transparency of access is maintained by cache coherency and NUMA hardware). Note also that although we do not exclude machines with more than one multicore CPU chip, these are not at all our main focus. We consider only problems that are tightly linked to multicore performance, and neglect a great many issues that are specific to multiple CPU machines.

The cores of the machine may support hardware multithreading; that is a single core may be able to execute more than one thread either by switching thread every machine cycle (classical multithreading), or by intermixing instructions from more than one thread in the execution pipeline (simultaneous multithreading — SMT). The degree of multithreading is generally bounded by a small constant. For example, each core in many Intel processors can execute up to two threads using SMT. This form of shared-memory multicore architecture encompasses almost all desktop, server and laptop systems, and a growing number of tablet, phone and embedded systems. Some important classes of systems that are excluded from our taxonomy include large scale parallel, distributed and cluster machines. We also exclude multicore processors with a distributed memory, such as the Cell BE processor, embedded DSP processors, and

[3]We do not consider transactional memory within our taxonomy. There has been a great deal of research on transactional memory over many years. But it is only since 2013, when Intel introduced the Haswell line of processors, that hardware transactional memory has appeared in a processor widely used in mainstream desktop machines. Our work is aimed primarily at mainstream software developers working on parallel software for popular multicore desktop, laptop and server computers. Transactional memory is likely to be important in the future, but at the moment it is simply too new in mainstream machines to draw conclusions from the experience of mainstream parallel software developers.

special-purpose network processors [71].

It is important to mention common concurrency problems such as deadlocks and race conditions [120]. As we have mentioned previously, a large proportion of programmers have to deal with them on a regular basis in their applications. While the diagnosis of race conditions and deadlocks are important problems[17, 46], we aim to focus the taxonomy exclusively on **performance** problems, as opposed to **correctness** problems. However, it should be noted that the boundary between correctness and performance in concurrent systems is fuzzy [12]. Deadlock is a good example of this, where it can be seen as both correctness issue and a degenerate case of lock contention where multithreaded execution cannot progress without external intervention, either automated or manual. We also focus the model on problems that have the potential to be diagnosed using data collected at execution time, and do not consider more abstract problems, such as high-level flaws in software architecture that hinder parallelization.

We include within our taxonomy a small number performance problems that are not unique to parallel software. In particular, we include performance problems relating to data locality, which arise in both sequential and parallel contexts. Data locality plays such an important role in program performance that it is impossible to ignore. Furthermore, a number of the performance problems that *are* unique to parallel software cause cache misses that might easily be confused with data locality problems. Thus we include locality problems, although at a lesser level of detail than performance tools for sequential programs, which might consider different types of locality problems (such as compulsory, capacity and conflict misses [71]) in more detail. For completeness we also deal briefly with some related problems such as page faults and input/output.

Finally, we note that we do not attempt to divide the problems according to type of application. Different types of applications often have common characteristics. For example, linear algebra computations typically operate on large dense matrices with very regular patterns of parallelism, whereas applications such as compilers have more linked data structures and irregular parallelism. However, a mapping between types of application and parallel performance problems is much less clear. For example, linear algebra applications also operate on sparse matrices, and the compact representation of these matrices is often irregular, unbalanced and requires synchro-

nization during updates. With the possible exception of the input/output category, none of these problems is unique to particular types of parallel applications. For example, one might be surprised to see poor load balance in an application performing simple, regular operations on dense matrices. But if such a load balancing problem exists, perhaps because of some simple mistake in the workload partition, the developer will want to know about it so that it can be fixed.

5.1.2 A TAXONOMY OF PARALLEL PERFORMANCE PROBLEMS

The taxonomy is comprised of seven broad categories, presented in the Table 5.1, with specific and distinct problems within each category. Below we describe each category and the rationale behind the problems included under them.

Category	Problem
Task granularity	Oversubscription Task start/stop overhead Thread migration
Synchronisation	Low work to synchronisation ratio Lock contention Lock convoys Badly-behaved spinlocks
Data sharing	True sharing of updated data Data sharing b/w CPUs on NUMA Sharing of lock data structures Sharing data between distant cores
Load balancing	Undersubscription Alternating sequential/parallel exec. Chains of data dependencies Bad threads to cores ratio
Data locality	Poor cache locality Poor TLB locality DRAM memory pages Page faults
Resource sharing	Exceeding memory bandwidth Threads competing for cache False data sharing
Input/output	Shared files Shared disk Shared network connection

Table 5.1 – Taxonomy of parallel performance problems.

1. **Task Granularity**. Task granularity refers to the number and size of the parallel tasks contained within the parallel program. In parallel programs it is often a

challenge to find enough parallelism to keep the machine busy. For example, Mak and Mycroft [104] study the limits on parallelism in several applications and find that parallelism is limited without very large changes. A key focus of parallel software development is designing algorithms that expose more parallelism. However, there are overheads associated with too many threads, and the cost of these overheads can exceed the benefits. The problems in this category deal with the overheads of starting, stopping and managing threads. Thus, the category might also be described as "thread management overheads".

2. **Synchronization**. Locks and other forms of synchronization are necessary to coordinate threads, but performance problems arise when threads spend too much time acquiring or waiting to acquire locks. Our focus is on multicore systems with a shared-memory programming model. Where data is shared and updated, some sort of synchronization is needed to ensure that all threads get a consistent view of memory. Even where there is no contention, the use of a synchronization primitive always causes some overhead. If the algorithm requires a large amount of synchronization, the overhead can offset much of the benefits of parallelism. Perhaps the most common synchronization mechanism is the lock; other mechanisms include high-level sychronization barriers and semaphores. Note that these synchronization mechanisms are built from low-level atomic instructions and *memory fences* (which enforce the order of memory operations), but these are typically hidden behind the interfaces of thread libraries, such as the well-known *pthread* and *Futex* libraries. Our category of sychronization deals with the overheads of acquiring, releasing and waiting for locks and other synchronization primitives.

3. **Data Sharing**. Data sharing problems can arise when parallel threads share the same data, and copies of the data must be passed back and forth between the parallel cores. Threads within a process communicate through data in shared memory. Sharing data between cores involves physically transmitting the data along wires between the cores. On shared memory computers these data transfers happen automatically through the caching hardware. However these transfers nonetheless take time, with the result that there is typically a cost to data sharing, particularly when shared variables and data structures are modified.

This category considers various overheads that arise under a number of different data sharing scenarios.

4. **Load Balancing**. Load balancing is the attempt to divide work evenly among the cores. Dividing the work in this way is usually, but not always, beneficial. There is an overhead in dividing work between parallel cores and it can sometimes be more efficient to not use all the available cores. Note that understanding many of the other performance problems requires an appreciation of the parallelization strategy, data dependencies and/or the parallel computer architecture. In contrast load balancing can be understood in relation to a much simpler measure of the amount of activity on each core. Within our taxonomy load balancing deals with trying to divide work evenly between cores whereas the closely related category of task granularity deals with the overheads associated with managing threads.

5. **Data Locality**. Data locality refers to the tendency for programs to reuse the same or nearby data repeatedly. For decades computers have relied on the principle of locality of reference; that is that if a piece of data is accessed it is likely that the same data, or nearby data in memory, will be accessed soon after. Problems with poor data locality are not specific to multicore, but it is impossible to talk about single or multicore performance without talking about locality. In the early 1980s a typical computer could read a value from main memory in one or two CPU cycles. However, between 1984 and 2004 processing speeds increased by around 50% per year, whereas the time to access DRAM memory fell by only 10%-15% per year. The result is that it now takes hundreds of processor cycles to read a value from main memory. This phenomenon is often called the "memory wall".

6. **Resource Sharing**. Resource sharing refers to multiple threads sharing the same physical hardware resource. Some novice parallel programmers expect a linear speedup: code running on four cores will be four times faster than on one core. There are many reasons why this is seldom true, but perhaps the most self-explanatory is that those four cores share and must compete for access to other parts of the hardware that have not been replicated four times. For example, all

cores will typically share a single connection to main memory.

7. **Input/Output**. Degradation of performance can occur when threads compete for I/O resources such as disk, file system or network[4]. While we include this category for completeness, as I/O can be very important to performance, it is not specifically a multicore problem, nor do multicore programs necessarily interact with I/O in complicated or unexpected ways. It is also heavily dependent on the software environment. For this reason, we do not include I/O problems in the analysis to follow, although it is an interesting topic.

We have compiled short descriptions and a brief analysis for each individual problem.

Task Granularity and Thread Management Overheads

Oversubscription. Oversubscription occurs when the work of the program is broken down into smaller tasks than is necessary to exploit the available parallelism [77]. Three cases can be considered: (a) the parallel program has more threads than cores; (b) the machine is running multiple applications and the number of threads exceeds the number of free cores; (c) multiple OS-virtual machines (VMs) are running on a physical machine, and the total number of threads across all VMs is greater than the number of cores (VM's are themselves a means to exploit multi-core infrastructure).

Oversubscription is often harmless and can even be beneficial if a large number of threads allows the cores to stay busy when some threads are waiting for synchronization or for other tasks to complete. However, oversubscription can become problematic if the overhead of managing or transitioning between threads becomes large. Our experience of teaching is that novice parallel programmers sometimes spawn a new thread (or sometimes two) for each level of recursion when implementing parallel versions of divide and conquer algorithms. This can quickly lead to very large numbers of parallel threads that compete for a limited number of processing cores.

Task start/stop overhead. This problem occurs where the amount of work performed by a task is insufficient to justify starting a separate thread to do it. The costs

[4]Many input/output performance problems in parallel systems arise from resource contention. Thus they arguably should be treated as resource sharing problems, rather than in a category of their own. However the enormous timescales of input/output (milliseconds, as compared to nanoseconds for many other performance issues we consider) and the additional bottleneck of much input/output passing through the operating system make these problems appear quite different to the developer.

of starting a thread are significant, so a sufficient amount of work must be done by the thread to justify it [23] (page 83). Programming systems such as OpenMP use a more sophisticated approach where they do not launch a new thread for each parallel task. Instead they start a pool of threads and put thread threads to sleep when they are not in use. This reduces thread start/stop costs by reusing a single thread for multiple purposes. However, even in these systems there is a cost from waking or putting a thread to sleep, albeit much lower than the cost of starting a new thread.

Thread migration. Thread migration refers to a thread moving from executing on one core to another. Each core has its own caches which contain data and code from the currently executing thread, and from threads that have executed recently. When a thread migrates to a different core, the benefit of this cached data is lost [36]. Similar issues arise with other state that is saved in each core relating to individual threads, such as information stored in the translation lookaside buffer (TLB). Where the number of threads fits within the number of available cores, threads will often stay on a single core for their entire execution. But when the number of threads is larger, we have idle threads waiting for a core to become available. There is a good chance that the first core to become available will not be the same as the last one the thread executed on. In such cases threads will tend to migrate from one core to another.

Synchronization

Low work to synchronization ratio. This problem occurs when the program synchronizes threads which do not perform enough work to justify the synchronization overhead. Acquiring and releasing a lock can be expensive [142]. Even if the lock is available, acquiring a lock generally requires an expensive "atomic" instruction, which both checks the lock and updates it in a single atomic step[5]. A lock is a small data structure in memory, so there may also be memory caching issues when acquiring

[5] A special case of this problem is so-called unnecessary locks. They are typically inserted into library code that might be called from parallel threads in a way that requires locks for correctness. However, when the library code is used in specific programs, the locks might be unnecessary for the specific context in which they appear. Thus, the library code repeatedly acquires and releases a lock that can never be in contention between multiple threads. Another variant of unncessary locks that can cause contention is where locking is overly-conservative. For example, the Python and Ruby interpreters' global interpreter lock (GIL) which guarantees that one interpreter thread is executing bytecodes within a process at any time [124]. The GIL was originally introduced to prevent race conditions in the Python memory manager, but it is widely regarded as being too conservative and a significant barrier to parallel performance.

a lock.

Lock contention. Lock contention occurs when a thread attempts to acquire a lock but the lock is already held by another thread [159]. In most cases where a thread attempts to acquire a contended lock, the thread must wait for the lock to be released before the thread can continue execution. Thus when locks are contended, threads are blocked from executing until the lock becomes free. Locks are generally used to protect shared data which may be updated. So if there is a lot of access to such data, or if a thread accessing the shared data holds the lock for a long time, then there will probably be a lot of contention.

It is also possible to use locks or other synchronization primitives such as semaphores to deliberately block the progress of threads. For example, it is common to have a master thread that generates tasks and places them in a queue, and a pool of worker threads that complete the tasks. When no work is available, locks or semaphores may be used to suspend the idle worker threads. The result is that it can sometimes appear that there is a great deal of contention between threads, when they are actually waiting for work to become available.

Lock convoy. Lock convoy [18] occurs under very specific circumstances when there are more threads than cores. On operating systems with pre-emptive thread scheduling, the execution time on the cores is shared among threads. If there are more threads than cores, only a subset of the threads are able to run at any time. A performance problem can arise if a thread holding a lock reaches the end of its allocated time slice and is therefore paused and put to the end of the run queue. If several other threads attempt to acquire the same lock, all will fail. When using standard locks (as compared to spinlocks) the operating system puts waiting threads to sleep. The waiting threads form a "convoy" behind the thread which holds the lock, but is paused. Putting each of the waiting threads to sleep and subsequently waking each thread takes time. If this pattern of locking occurs repeatedly, the overhead can be significant.

Badly-behaved spinlocks. When a spinlock is already locked, all other threads that attempt to acquire the lock go into a loop waiting for the lock to become free, which can result in useless spinning around the loop. When attempting to acquire a lock, a thread will check the lock to see whether it is available. In common imple-

mentations of locks, such as those found in the `pthreads` library, a thread that fails to acquire a lock is suspended by the operating system until the lock becomes available. Spinlocks are a special type of lock that does not suspend waiting threads [10]. Instead the waiting thread repeatedly tries to acquire the lock until it becomes available.

If the lock becomes available soon, then spinlocks are usually much faster than standard locks. If the lock continues to be held for significant time, then the waiting threads occupy cores but make no progress. Note that if the thread holding the lock exceeds its execution time-slice and is preempted by the operating system, other threads can spin uselessly waiting for the lock-holder to next execute and release the lock. These waiting threads occupy cores that might otherwise be available to excute the thread that holds the lock, and ultimately allow it to release the lock. As a result, spinlocks can lead to catastrophic slowdowns if they are heavily contended.

Note that it is possible for a careful programmer to make the worst case performance of spinlocks extremely unlikely. The developer must be careful to keep the number of executing threads no greater than the number of available cores, to reduce the chance that a thread holding a spinlock will be preempted and continue to hold the lock while waiting to get to the top of the runqueue. However, limiting the total number of threads is difficult on a multicore machine that can execute more than one program.

Data sharing

True sharing of updated data. This problem occurs when the same variable is written/read by different cores. When a core writes to a shared variable, the cache coherency hardware invalidates all other copies of that variable in other cores' caches [129]. As a result when the other cores next attempt to read the variable, they will discover that the copy in their cache is invalid. The cache hardware will then fetch the latest copy of that variable from wherever it resides in another cores cache, a shared cache, or main memory. This is known as a cache coherency miss, and the time taken is similar to other types of cache miss. When shared data is updated a lot, there will be many coherency misses.

Sharing of data between CPUs on NUMA systems. This problem occurs on multiple CPU machines, which often have non-uniform memory access times for different

CPU chips; when memory is shared between threads on different CPUs, some of the main memory accesses will be to non-local main memory. When a linked data structure is constructed by multiple threads, different parts of the data structure may be in different local main memories. The caching and coherency system will ensure that all threads see the correct values, but physically moving the data between CPUs takes time [19].

Sharing of lock data structures. This problem occurs when locks are alternately acquired by different threads, and the data structure containing the lock must be repeatedly transfered bewteen cores. Locks consist of data structures in memory and code to acquire and release the lock [136]. The code must use special atomic machine instructions to ensure mutual exclusion around the lock. However, the lock data structure is stored in normal memory locations. For the simplest spin locks, the lock data structure may consist of a single boolean variable. When locks are acquired and released, the lock data structure is modified. The lock data structure is shared among the threads that acquire and release the lock, and therefore exhibits the same behaviour as any shared data structure that is updated by multiple threads. Problems thus occur when a large number of locks are acquired or released, as when any shared data structure is updated by more than one thread.

Sharing data between distant cores. This problem occurs when data is shared between cores and must physically move between the cores when it is updated [178]. On the recent generations of mainstream Intel processor (Skylake, Haswell, Ivy Bridge, Nehalem) each core has had its own L1 and L2 cache. In contrast on some earlier Intel multicore processors (Core, Penryn) had a shared L2 cache for each pair of cores. With these configurations, some cores are "closer" than others, in the sense that the cost of sharing data with another core that shares an L2 cache is much lower than sharing data with a more "distant" core that is part of a different L2 cache cluster. On all recent mainstream Intel multicore processors the L3 cache is shared by all cores.

Note that sharing data between distant cores is a special case of true sharing of updated data. In both cases, the source of the problem is the same: there are multiple copies of shared data in the caches of each core that uses that data. Before the data can be updated in one cache, all other copies must first be invalidated. When the data is next used by another core, that core must again load that data to the cache. The

special case of data sharing between distant cores is different to the more general case of updating shared data in two respects. First, when cores are distant, the impact of transferring data over a long distance is large. Second, there is a good solution to the problem of sharing data over long distances: change the mapping of threads to cores so that communicating threads are located closer together. This does not remove the need to move data between cores, but it reduces the distance that the data must travel.

Load balancing

Undersubscription. Undersubscription occurs when there are too few threads actually running on a particular machine, resulting in unused cores [69]. Where the program contains sufficient parallelism to usefully exploit the additional cores, the result of undersubscription is that the program takes a longer time to execute. Sometimes a parallel program is heavily optimized for a particular machine, and the number of threads is hard-wired into the program specifically for that machine. When the program is executed on another machine with a larger number of cores, the additional cores remain idle. Problematic undersubscription presupposes that it is both possible and profitable to execute more threads.

Alternating sequential/parallel execution. This problem occurs when a program passes through successive sequential and parallel phases, such as fork-join orchestration models, and the sequential part slows down the program [9]. As originally formulated Amdahl's law divides program execution time into sequential phases and parallel phases, with performance limited by the sequential part. Even if you have infinite processors and the parallel part can be sped up infinitely (meaning the execution time of the parallel part approaches zero), the maximum overall speedup is limited by the sequential part. Of course real programs are more complicated than Amdahl's law suggests. Few programs scale linearly with large numbers of processors, and if the same effort is applied to optimizing the sequential code as is applied to parallelizing the parallel code (unless the sequential code has already been optimized), it may be possible to improve its speed with algorithmic and coding changes. Changes to the orchestration model may also help remedy such situations.

Chains of data dependencies, too little parallelism. This problem occurs where a thread is waiting for a result produced by another thread so that it can continue

computing [171]. There are many well-known parallel programming patterns that exhibit this problem. For example, recursive divide and conquer algorithms such as quicksort can be easy to parallelize. But the parallelism is usually at the lower levels of recursion, and the higher levels have much less parallelism. Similarly, pipeline type parallelism – such as performing different stages of image processing on different cores – can be a good parallelization strategy for stream-like processing, but it is easy to get the balance between the cores wrong.

Bad threads to cores ratio. This problem occurs when the work is divided into chunks not matching appropriately the number of cores. We normally think about trying to keep the number of threads equal to the number of cores. But sometimes we divide the parallel work into a set of parallel tasks of roughly equal size. If we assign each of these tasks to a thread then the load balance depends on how the the number of tasks relates to the number of available cores. If the number of tasks is an even multiple of the number of cores, then the load balance will usually be reasonably good. For example, if there are eight tasks and four cores, then each core will perform two tasks, and the total time will be roughly the amount of time needed to perform two tasks on a single core. In contrast, nine tasks would take much more time, because the last would execute alone [97].

Data locality

Cache Locality. This problem occurs when the data is not present in a reasonably nearby location, resulting in more distant cache or main memory fetches [84]. When accessing memory via a cache, the cache will check whether that data is already in cache by inspecting the tags in the cache. Caches do not fetch single values from main memory. Instead they bring in a full line of data, which on most machines is 64 bytes. Each cache line is aligned on a 64-byte boundary, and the cache keeps track of whether each line has been read only (clean) or whether it has also been written to (dirty). When a clean cache line is evicted from the cache it can simply be discarded. When a dirty cache line is evicted, it must be written out to the next level of cache or to main memory.

Cache misses arise in both sequential and parallel programs as a result of poor data locality, and are therefore not specific to multicore performance problems. However,

cache locality is central to the performance of modern multicore systems, and competition between threads for limited cache space can greatly exacerbate data locality problems within each thread.

As we describe in our section on performance problems relating to data sharing, parallel programs have an additional source of cache misses known as *coherency misses*. Locality and data sharing performance problems have very different causes and solutions, so it is therefore important that developers can distinguish between the two.

TLB Locality. Translation lookaside buffer (TLB) locality problems occur when the program references a large number of pages of memory [82]. Almost all modern operating systems support virtual memory, allocating memory in fixed sized "pages" of perhaps 4KB. These pages can be moved around in memory, or swapped out to disk. Because pages can move around, the operating system needs to keep a table to map between the addresses that the program uses (virtual addresses) and pages of real memory (physical addresses). Modern processors provide a special cache to store the most frequently used parts of this table, known as the translation lookaside buffer (TLB). The TLB relies on most memory accesses referencing a small number of pages (a form of locality). If the program instead references many pages, there will be TLB misses.

DRAM memory pages. This problem occurs when the memory accesses are not targeting the same physical DRAM pages [41]. This is a slightly obscure problem, but physical DRAM is also divided into "pages" of perhaps 4 or 8KB. Successive memory accesses to the same page are faster than accesses to different pages.

Page faults. This problem occurs when the program uses more memory than is physically available on the machine [50]. The operating system keeps excess memory pages on disk. When the program attempts to access an address that is stored on disk, the CPUs memory management unit generates an exception known as a page fault. This causes the operating system to discard (if clean) or write out to disk (if dirty) one or more pages of memory, and read the required page(s) from disk into memory.

Resource sharing

Exceeding memory bandwidth. Memory bandwidth problems occur when the memory bus is saturated with requests [103]. On almost all current multicore processors

external DRAM main memory is shared between all the cores. To access main memory, a core must gain access to a memory bus that is shared by all cores. A single core with very poor locality can easily generate enough memory requests to occupy the majority of the time on the memory bus. When four, eight or sixteen cores are active on a single CPU, competition for access to the memory bus can become intense, and cores can spend a lot of time waiting for memory requests on the highly-contended bus to return.

Competition between threads sharing a cache. This problem arises when a thread loads data that displaces existing data belonging to another thread that shares the same cache. When two or more threads run on cores that share the same cache, the threads may operate on the same data or separate data. If they share the same data, then all threads benefit from the cached data [86]. However, when each thread operates on separate data there may be insufficient space for each thread's data. The result is competition between threads for space in the shared cache. The data being used by thread A may displace other data being used by thread B. Zhuravlev et al. describe these as "contentious threads" [179].

One solution to this problem is to attempt to map threads to cores, so that threads which operate on the same data are mapped to the same core. Operating systems such as Linux and recent versions of Windows allow threads to be mapped to particular cores using *processor affinity*.

False data sharing. This problem occurs when a cache line is invalidated on a core due to another core writing to it, but the threads aim to read/write different variables [161]. The cache coherency system is responsible for ensuring that when multiple copies of the same variable are present in different caches, that the different copies are kept coherent. Common coherency protocols solve this problem by requiring any core that writes to a cached variable to first ensure that it has the only copy of that variable. This is achieved by invalidating all other copies of the variable before the write is allowed to proceed. It is important to note that cache coherency is done at the level of cache lines, not individual variables. Thus if several variables occupy the same cache line, writing to any one of them will invalidate all copies of the cache line in the caches of other cores. Thus, it is possible to cause cache invalidation (or coherence) misses even without writing to a shared variable; it is enough that two variables share

the same cache line. This is known as false data sharing.

RELATIONSHIPS BETWEEN PROBLEMS

As noted at the beginning of this section, it is not always clear whether to place a particular problem in one branch of the taxonomy or another. False sharing arises because several variables, each of which is not shared, can be mapped to the same line of a cache. Arguably the line of the cache is a physical resource that is shared by variables from different threads. However, the problem is also linked to the location of data in memory, which is a feature of locality problems.

In fact, three of the major categories of problems are interconnected: data sharing, data locality and resource sharing. All three deal with the patterns of access to data in memory within and between threads. However, there is a helpful distinction between the problems. "Data sharing" is about the locality problems that can arise when sharing data between threads. The "data locality" category is primarily about data access patterns largely independent of sharing between threads. Finally, the "resource sharing" category relates to the problems that arise when multiple threads compete for the same physical hardware resources for access to data.

5.1.3 PROBLEM IMPORTANCE

Given the range of problems within our taxonomy one might ask how important each problem is. For example, a pagefault is perhaps of the order of 100,000 times more expensive than an L1 cache miss, so does that mean that page faults are more important than cache misses?

The impact of a performance problem is related to (1) how often during execution that problem arises, (2) the performance cost each time the problem arises and (3) how much time the parallel program spends doing other things. Context will likely play a significant role. For example, in the parallel programs we have worked with L1 cache misses are extremely common, whereas page faults are rarer: typically a large multiple more than 100,000 times rarer. So in our own case, L1 cache misses are a much more important problem than page faults. Similarly, the worst case cost of badly behaved spinlocks can be much greater than the cost of simple lock contention, but the latter is

by far more common in the programs we have worked with.

One of the goals of our study is to obtain a broad characterization of how frequently different performance problems arise to the extent that they have a significant impact on performance. Without preempting the more detailed discussion of these results, the data in the Figure 5.4 suggest cache locality is indeed more commonly a significant performance problem than page faults. And simple lock contention is much more frequently a significant performance problem than badly behaved spinlocks.

A related question is whether a taxonomy should be aimed at higher-level performance questions. The choice of data structure or algorithm usually has a much greater impact on performance than its parallel implementation. For example, for large n even a sequential $O(n^2)$ algorithm will usually be much faster than a parallel $O(n^3)$ algorithm. Certainly, in some cases, the appropriate course of action when faced by poor performance is to look for a better algorithm. On the other hand, when developers implement their parallel algorithm on a real multicore computer, they often encounter strange performance behaviour. For example, false sharing can have a huge impact on performance, but as will we be seen in the next section, our data suggests that it is not widely known or understood.

5.2 PROBLEMS IN THE WILD

Before investigating the information required to diagnose these problems, we felt it would also be valuable to examine:

1. Whether developers are familiar with the problems listed within the taxonomy.

2. Which problems are more frequently encountered.

To investigate these questions, we performed a survey with a broad range of developers. This survey presented the list of problems given above (with descriptions), and the participants could specify whether they are familiar or not with a problem, and if they are familiar with it, how often they encounter it in their daily work. As the results of this survey are of interest when discussing the expert validation exercise, we present it here first.

5.2.1 METHODOLOGY

The survey was designed to assess and validate the list of parallel performance problems within a larger sample of developers across industry and academia. The survey was designed to be administered through a publicly available online interface and take around 10-15 minutes to complete.

Recruitment was primarily conducted through LinkedIn professional groups where we posted advertisements, augmented with calls issued through personal social networks. A variety of different LinkedIn discussion boards were used, ranging from small and niche such as "Multicore & Parallel Computing"? to more general such as "Java Developers"?.

The survey was designed to simultaneously evaluate two dimensions, for each problem that is present in our taxonomy:

- **Familiarity**. We wanted to know which problems developers are familiar with and which ones are more exotic and unfamiliar to a general community of programmers.
- **Frequency of Diagnosis**. We wanted a broad indication of how often particular problems get diagnosed by programmers, on a relative scale.

Participants were presented with a list of parallel performance problems with descriptions of each and they were asked to indicate whether they:

- *Are familiar with the issue (have heard of this problem)*
- *Have never encountered the issue*
- *Have encountered the issue once*
- *Have encountered the issue occasionally (e.g.: 2 or more times over past few years)*
- *Have encountered the issue frequently (e.g.: several times per year or per project)*

The questions are intended to focus on the participants own systems. By design of the survey, to answer "never", "once", "occasionally" or "frequently", the participant had to first select that they are familiar with the issue (i.e. either encountered it themselves or just heard about it). By extension, if the participant expressed familiarity

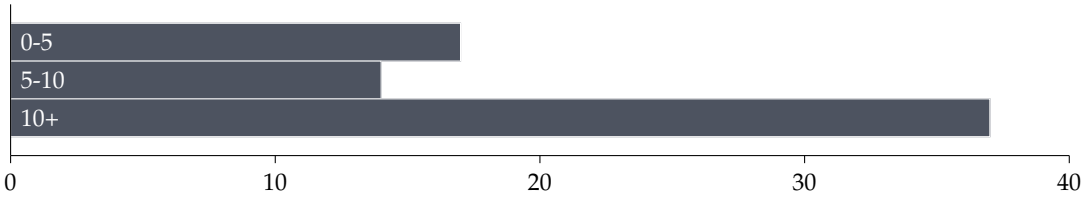

Figure 5.1 – The professional (years of) experience distribution of the developers who participated in the problem frequency/familiarity study.

but selected "never encountered", this would suggest that the participant had heard about the issue, but never actually encountered it in their own systems.

We also collected demographic information about the participants, such as age, gender, highest level of education, years of professional experience and a self-assessed expertise in the field of parallel programming. Participants could skip questions if they wished and were invited to give remarks or report missing performance problems.

5.2.2 RESULTS

In total, we had 71 participants, mostly professional programmers and some experts in multicore and distributed computing with a wide range of expertise. The most frequent age category for participants was 26-35, with 24 participants in this range, with 36-45 the second most common category with 23 participants. Gender balance was poor, with 69 male and 2 female. For highest level of education, the most common category was Masters level with 33 participants.

Figures 5.1 and 5.2 present distributions of experience and self-assessed expertise respectively of the participants in the survey. As can be seen, the survey attracted programmers with 10 or more years of experience (see Figure 5.1) in the field. Moreover, almost half (32 participants) not only have 10+ years experience, but also have assessed themselves as having above-average expertise in parallel programming. While this might mean that the sample is weighted towards more expert programmers, it also means the participants have enough experience to have had a chance to encounter a range of problems.

To enable richer consideration and discussion, the data in Table 5.2 will be discussed in more detail in Section 7.4 together with the study presented in the next

Figure 5.2 – The distribution of self-assessed expertise of the developers who participated in the problem frequency/familiarity study.

section. However, we note at this point that many of the problems are encountered at least occasionally by a substantial proportion of developers. Lock contention was both familiar to the highest proportion of programmers, and most frequently encountered. Task start/stop overhead was also a very familiar problem, but less frequently encountered. Lock convoy was the least familiar to programmers. Familiarity is a prerequisite for diagnosis, and so this must be taken into account when interpreting these figures.

Problem	Unfamiliar	Never	Once	Occasionally	Regularly
Task granularity					
Oversubscription	30%	9%	10%	39%	12%
Task start/stop overhead	9%	23%	14%	35%	19%
Thread migration	25%	25%	14%	25%	11%
Synchronisation					
Low work to synchronisation ratio	22%	22%	9%	29%	18%
Lock contention	5%	14%	8%	34%	39%
Lock convoy	62%	10%	6%	16%	6%
Badly-behaved spinlocks	38%	21%	8%	23%	10%
Data sharing					
True sharing of updated data	25%	15%	2%	37%	22%
Sharing of lock data structures	33%	17%	7%	28%	15%
Sharing data between distant cores	40%	20%	10%	15%	15%
Load balancing					
Undersubscription	37%	14%	7%	22%	20%
Alternating sequential/parallel exec.	16%	10%	9%	34%	31%
Chains of data dependencies	11%	23%	7%	27%	32%
Bad threads to cores ratio	20%	24%	10%	36%	10%
Data locality					
Poor cache locality	10%	22%	5%	27%	36%
TLB Locality	37%	20%	14%	19%	10%
CPU data sharing on NUMA	38%	13%	10%	28%	11%
DRAM memory pages	48%	19%	5%	17%	10%
Page faults	24%	15%	8%	39%	14%
Resource sharing					
Exceeding memory bandwidth	21%	17%	14%	17%	31%
Threads competing for cache	19%	20%	15%	27%	19%
False data sharing	41%	17%	8%	24%	10%

Table 5.2 – Familiarity and frequency for performance problems. Participants who stated that they encountered 'never', 'once', 'occasionally' or 'regularly' also stated that they are familiar with the problem.

5.3 OBSERVATIONAL MODEL

This section presents an Observational Model designed to serve as a link between: (a) **concrete data we can measure or calculate** (e.g.: operating system events, hardware performance counters or other instrumentation) and (b) **parallel performance problems** presented in section 5.1.

The intention is not to provide a definitive or mathematical model, but rather a useful conceptualization that can be used by tool developers building performance analysis tools such as interactive visualizations or performance prediction algorithms.

To link concrete data with more abstract, often not precisely defined, performance problems where some degree of subjective judgement must be made, we have based the model on observations. For example, consider when a developer observes some performance data and concludes that *"a high number of cache misses are generated by the program"*. This is what we term an observation. The developer then draws some conclusions, for example that particular observation could mean that there is a data locality performance problem in the program.

We define two categories of observations: **indications** and **contra-indications** of problems:

- **Indication** or **Strong Indication** of a performance problem means that that observation O_i implies that a particular problem, say problem p, might be present in the program P.

$$Indication = O_i \Rightarrow p \in P$$

- **Contra-indication** or **Strong Contra-indication** of a performance problem, on the other hand, means that that observation O_i implies that a particular problem is more likely to **not be be present** in the program P.

$$Contraindication = O_i \Rightarrow p \notin P$$

The observations themselves are short phrases, describing a **measurable or calculable** event, for example: "high number of L1 cache misses" or "number of threads is larger than number of cores". It is important to note that the observations do not con-

tain any specific numbers or even percentages but instead contain subjective words such as *high* or *low*. The rationale behind this is that the development context will determine the thresholds for a particular value being *high* or *low*. For example, applications that operate on dense matrices usually have much better data locality than those operating on sparse matrices. Hence the threshold for *high* and *low* can depend on the type of application. They would also vary depending on the organizational context, the available resources, the target performance, etc.

At the present time, there are hundreds of performance counters available on a typical commodity-hardware CPU chip, hundreds more off-chip, and thousands of different Operating System events. It is important to note that even for what might seem relatively straightforward performance data (such as cache misses), some form of calculation involving several performance counters is often involved (e.g. summing different types of miss), and to obtain useful data at thread level, these will need to be correlated with Operating System events or program instrumentation.

The counters and metrics we use are just a tiny fraction of those that are available on modern multicore computers and operating systems. An obvious question is how we select from among the thousands of potentially useful measures. In fact, the great majority of measures are focused on very low-level details of the computer architecture or operating system. Few of these measures are aimed at application programmers, and even fewer at providing useful information about multicore execution. We selected those that seemed to have the potential to identify multicore performance problems. Note that modern performance analysis tools, such as Vtune [137] and TAU [146] use heuristics to map many of these measures to particular threads and lines of source code of a parallel program. This mapping from measures to source code is invaluable when relating performance problems to particular parts of the parallel program.

5.3.1 CROSS-VALIDATION

Some components of the model are straightforward and would be expected to hold in most or all development contexts. Other aspects may be more controversial or dependent on development context, or may require that additional data be examined

simultaneously. To identify these different components, we performed a study with 10 experts in the parallel programming domain. While inter-rater agreement studies are usually conducted with a small number of experts (2-3 typical), the complexity of the domain motivated a larger number of experts. Each expert was presented with a series of problems and observations related to each problem and had to annotate whether an observation is either: (a) strong indication, (b) indication, (c) contra-indication, (d) strong contra-indication of a particular problem or (e) is irrelevant to a particular problem, using a standard Likert-style scale. The experts were able to skip observations or problems they were not familiar with and add missing observations.

The participants for the inter-rater study were recruited through the means of chain-referral sampling (also known as snowball sampling: recruitment of participants is done both directly and through recommendations from existing participants). This sampling method was intended not only to find highly motivated participants for the inter-rater experiment but also filter out non-experts by having "experts recommending other experts". The experts recruited included developers with a background in high-performance computing, parallel software for web services, parallel software for games consoles, and parallel software for the desktop. The study was delivered in the form of an on-line web interface which standardized the administration of the materials and allowed it to be conducted remotely. The duration of the experiment varied from one participant to another, as would be expected given the relatively complex and demanding task, and on average it took the experts about 40 to 50 minutes to complete everything.

Each expert was asked to annotate the same 110 observations presented in the validation. The 10 experts made 933 annotations while skipping 167. Most of the observations (85%) were annotated by the experts. Skipped observations were not included in the analysis. Due to the nature of the data, skipped questions are not expected to impact on the results in any significant way; the purpose of the study is expert validation, and hence only answers that the experts are confident in are of interest. Note that there is an important distinction between a skipped answer and a neutral response (that the observation is irrelevant to the problem).

We then performed an inter-rater agreement analysis consisting of two parts:

1. Calculation of inter-rater agreement value to generally validate the viability of

the model. A high level of agreement between expert annotations of an observation supports the validity of that aspect of the model.

2. Creation of visual displays as illustrated in the Figure 5.3 to help visualize the details of the results and identify observations that are most promising for performance problem diagnosis, or contentious issues requiring further investigation.

We performed the calculation of inter-rater agreement using Fleiss' kappa, a commonly accepted statistical measure for assessing the reliability of agreement among multiple raters (i.e. 10 experts). This measure is more robust than simple raw (percentage) agreement, as it takes into account the situations that could be expected by chance. The kappa scores for the agreement of annotations of indications and contra-indications were respectively:

$$k_{indication} = 0.383$$

$$k_{contraindication} = 0.529$$

While these kappa values can be interpreted, according to Landis and Koch [91] as fair and moderate agreements respectively, we need to examine the agreements for each observation individually to see which ones are agreed upon, as one of the goals of the study is to identify which indications and contra-indications have high and low levels of agreement. Many individual indications and contra-indications have unanimous or near-unanimous agreement, whereas opinion on others is divided. It is interesting to note that experts seem to agree on which observations are contra-indications of performance problems more strongly (higher kappa score) than on indications (lower kappa score).

One possible interpretation of this is that it may be due to "necessary but not sufficient" conditions. The presence of a problem may be more difficult to agree upon, as some symptoms may have multiple causes, or there may be multiple variants of a given cause (e.g. L2 vs. L3 misses). For example, a high number of L2 misses might indicate a data locality problem but does not guarantee that one is there. On the other hand, a contra-indication can tell us that a problem is not there. For example, a low number of cache misses effectively rules out data locality problems. However, we should be careful about adding too much weight to this discrepancy as the model

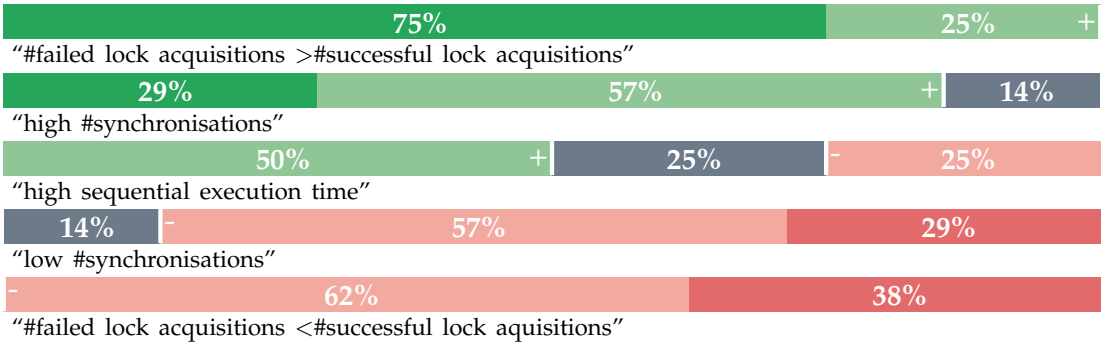

Figure 5.3 – Levels of experts' agreement on observations related to the "**Lock contention**" problem.

puts forward more indications than contra-indications.

Figures 5.3 and 5.5 to 5.13, illustrate various observations for the problems we validated and present the information as a sorted horizontal stacked bar charts, the strongest indication on top and strongest contra-indication on the bottom. The dark and light green segments of the graphs (boundary annotated with "+") indicate strong indication and indication, grey sections neutral, and red segments a contra-indication (boundary annotated with "-"). Graphs containing both red and green signify areas of disagreement, whereas graphs containing a single colour (red or green) signify agreement. For example, in the Figure 5.3 the observation labeled as "#failed lock acquisitions > #successful lock acquisitions", presenting the situation where we observe that a number of failed lock acquisitions is larger than number of successful lock acquisitions, was rated by most experts as being a strong indication of the lock contention problem being present. On the other hand the contrary observation "#failed lock acquisitions < #successful lock acquisitions" was rated as a contra-indication (but not a very strong one) of a lock contention problem.

In the next section we discuss the results of this study, together with the results of the survey presented in section 5.2.

5.4 DISCUSSION

In this paper, we present a taxonomy and model that we hope will have practical implications and can be relatively easily understood and applied. The two studies presented in the previous sections provide a number of interesting topics for discussion

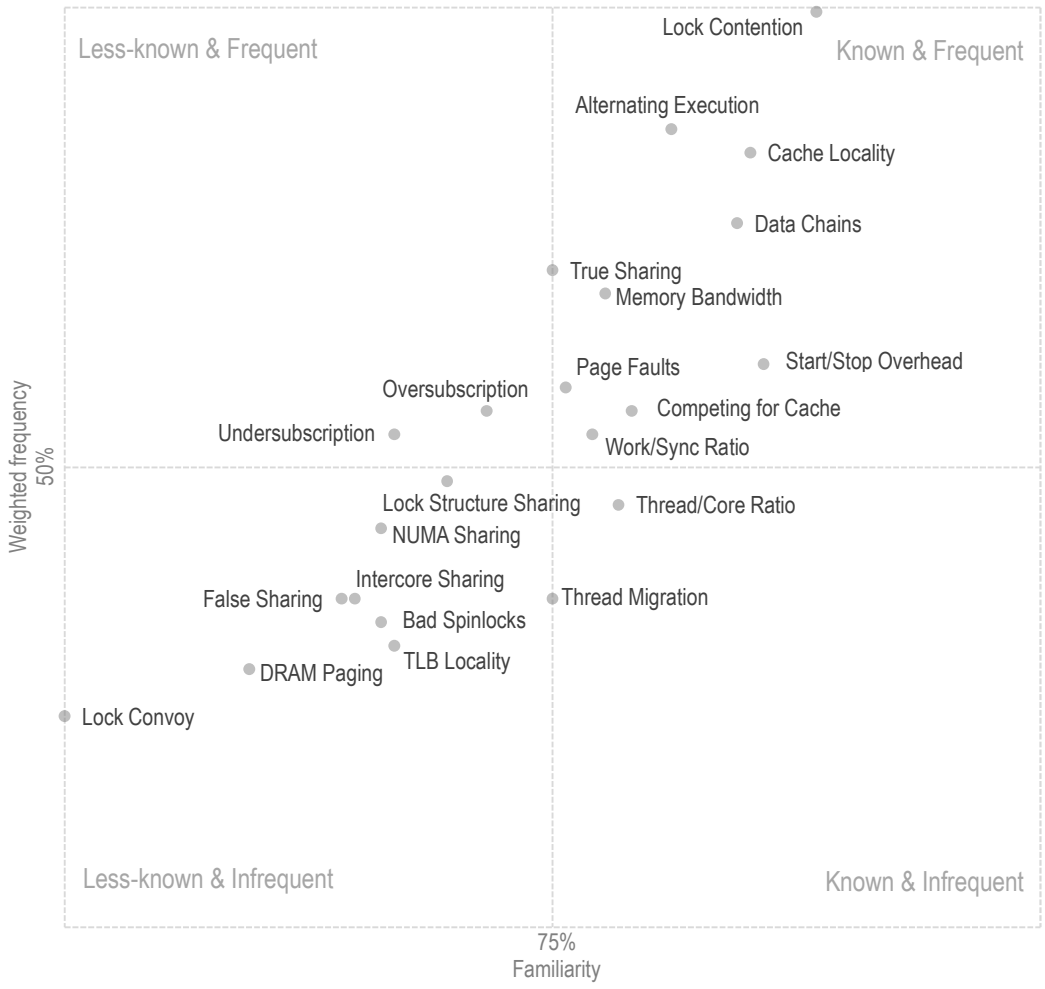

Figure 5.4 – Quadrant plot of parallel performance problems mapped against Familiarity and Frequency.

with respect to the model and taxonomy.

Figure 5.4 shows a plot of the familiarity of each of the parallel performance problems to participants, and how frequently they see them in practice. The plot is based directly on Table 5.2. Weighted frequency is on the y axis, ranging from 0 to the maximum represented by lock contention (midpoint 50% of maximum). Familiarity ranges from 38% to 95%, with the median of 75% used as the midpoint. Table 5.2 contains more nuanced data, but the figure provides a quick overview.

Very broadly, one can see that the problems cluster around a diagonal band from bottom-left (*less-known and rare*) to top right (*known and frequent*). Thus, problems which are more familiar to developers are also more frequently observed by them. Problems which are less familiar, are also observed rarely (in general). This suggests

that the direction of causality between participant familiarity with a problem and the frequency with which they see it in practice is (in general) that developers become more familiar with problems that occur often.

The "Less-known and Frequent" quadrant is counter-intuitive, and relates to problems that are not familiar to many programmers, but frequently encountered by those programmers who are familiar with them. While there is a likely causal relationship between the two (a programmer might become familiar with the problem because they work in a domain where they are likely to encounter them), closeness to this quadrant might also emerge due to the problem being difficult to diagnose.

Looking at the problems in the *less-known and rare* quadrant, some are quite technical problems. For example, identifying problems with DRAM paging and TLB locality require quite a low-level understanding of parallel computer architecture. Similarly both lock convoy and problems with badly behaved spinlocks arise when a thread holding a lock is descheduled by the operating system. These problems require a good understanding of how locks are implemented at a low-level, some understanding of operating system scheduling, and a good ability to reason about the impact on other waiting threads that can arise from the thread holding the lock being descheduled. This makes these types of problems both difficult to understand in principle, and difficult to identify in practice. A remark left by one of the particpants illustrates the latter point:

> Participant: "*Even though I am familiar with many of the concepts above in theory, it has been difficult to diagnose in code what a performance bottleneck could have been attributed to, and therefore to know if I had encountered it or not. In a professional environment there are constraints also to my time, so that I often have to submit code that is simply "good enough" where I have not deduced all problems.*"

On the other hand, the problems that are identified as known and frequent are more closely related to the parallel program itself and its orchestration model. These include lock contention, alternating parallel and sequential execution, and chains of data dependences. The two major exceptions to this pattern are cache locality and memory bandwidth problems, which arise from interactions with the parallel hardware rather than being part of the program.

5.4.1 FAMILIAR AND FREQUENT PROBLEMS

First let us examine the problems that are often diagnosed by the developers we surveyed and that they are familiar with.

Lock contention (Figure 5.3). As one would expect, lock contention is a well known and frequently encountered problem. There are solutions and various different ways to diagnose the problem (e.g.: [75, 159]). Experts agreed that the strongest indication of lock contention is a simple proportion between the count of failed lock acquisitions and successful ones. Experts also agree that a high number of synchronizations is an indicator of the problem and a low number is a contra-indicator.

However, there is significant disagreement about whether a high level of sequential execution is an indicator or contra-indicator of lock contention. During the original development of the model, we identified a high level of sequential execution time as a likely indicator of lock contention, as it correlates with threads being serialized by lock contention. However, 25% of experts disagreed. And indeed they are correct that if the program is mostly inherently sequential then we we may not see much lock contention because at least two parallel threads are needed for contention. A more correct refinement of the model might state that if there are many unsuccessful lock acquisitions *and* a great deal of sequential execution time, then it is likely that parallelism is being severely limited by lock contention.

A recent paper on lock contention goes into more detail on different strategies for gaining insight into the performance impact of various locking mechanisms, including spinlocks; numbers of successful/unsuccessful lock acquisition attempts play an important role in the strategies presented [159].

Poor cache locality (Figure 5.5). This problem seems similar to lock contention in terms of its importance and frequency of diagnosis. The well-known paper "What Every Programmer Should Know About Memory" by Ulrich Drepper illustrates the problem with a simple matrix multiplication program [41]. This problem can be diagnosed by using low-level hardware performance counters[6] which can be accessed via tools such as Intel VTune or Intel Performance Counter Monitor, for Intel-family

[6]Hardware performance counters, are a set of special-purpose registers built into modern microprocessors to store counts of low level hardware events. These events include cache misses, branch mispredictions and instructions executed. They were originally added by hardware designers to help them understand bottlenecks in the hardware, but are now used for software performance analysis and tuning.

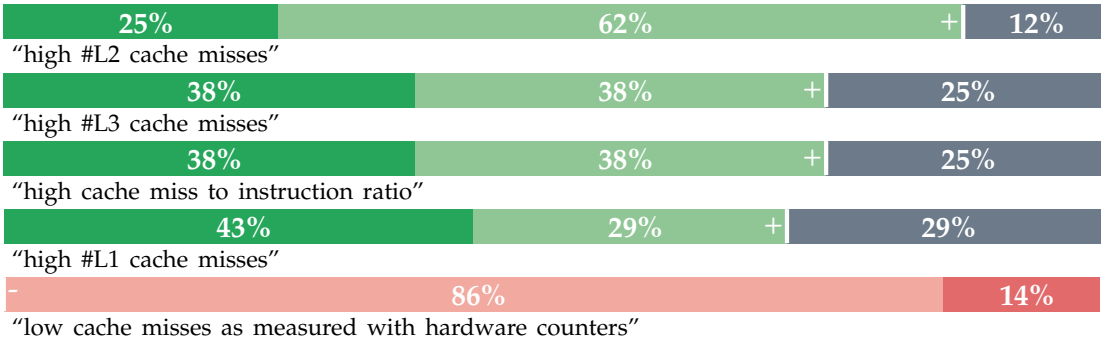

Figure 5.5 – Levels of experts' agreement on observations related to the "**Cache Locality**" problem.

CPUs. According to the model, cache locality problems can be identified by looking at level 1, 2 and 3 cache miss hardware performance counters and a program that contains low cache misses would be less likely to have this performance problem.

While this may seem obvious, the experts were not entirely in agreement on the strength of the indications. For example, around 71% of experts agreed that a high number of L1 cache misses was an indication of a cache locality problem, but fully 29% were not convinced that high L1 misses necessarily means that there is a cache locality problem. A similar pattern arises for L2 and L3 misses, albeit to a lesser extent. In other words, some experts regard L1 misses as the "real" cache locality problem, others are more focussed on L2 or L3. When one considers that L1 misses are typically much more frequent than L3 misses, but the cost of L3 misses is much higher it is easy to see how this disagreement might arise. For applications with relatively small working sets, L3 misses may be so rare as to be negligible, but it is easy to have poor locality within a small working set and experience a great many L1 misses. On the contrary, in applications with many L3 misses, even large numbers of L1 and L2 misses may seem insignificant. Indeed when we inspected the data in more detail we found different experts focusing on different levels of cache misses.

Alternating sequential and parallel execution (Figure 5.6). This particular problem is related to the way parallel programs are commonly structured, with large portions of single-core execution (sequential phase) and parallel execution on multiple cores (parallel phase). This often occurs in common design patterns for parallel programming, such as the *fork-join, scatter-gather, map-reduce* patterns. There is a great deal of agreement among our experts on indicators of the problem, which suggests

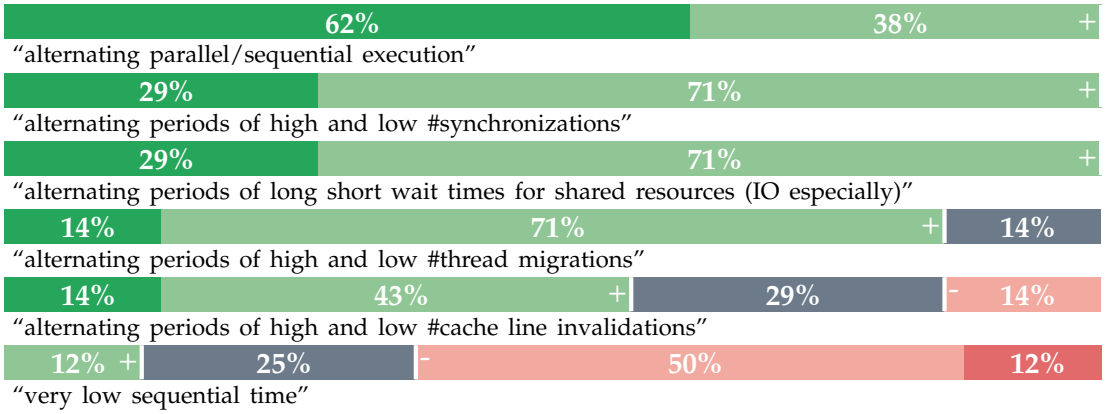

Figure 5.6 – Levels of experts' agreement on observations related to the "**Alternating sequential/parallel execution**" problem.

that with tool support it can be reliably diagnosed.

There is some disagreement among experts on whether a varying number of cache line invalidations indicates that the problem exists. In parallel programs cache line invalidations are almost exclusively the result of shared data being updated; before a copy of the data in one cache is updated, all other copies must first be invalidated by the cache coherency hardware. Updating of shared data happens only during concurrent execution, but the absence of shared updates does not guarantee the absence of parallel execution. Parallel threads may simply update independent data, with little updating of shared data. Therefore this is a much weaker indicator than some other measures, and it makes sense that the experts show some ambivalence about it. Fortunately, there are other much stronger indicators that can be used to identify the problem. So we conclude that the problem is frequent in practice, and that experts agree that is can be diagnosed with a small number of metrics.

Chains of data dependencies with too little parallelism (Figure 5.7). This is another example of a performance problem that occurs in standard parallel programming patterns, such as recursive divide and conquer algorithms.

In interviews carried out for a previous study [12], this was found to be a difficult issue. The model does not provide clear indications to reliably support identification of this problem as the expert assessments were in disagreement. Our data is very clear that the problem is both well-known among parallel software developers, and frequent in practice. But experts are not in agreement about how it might be identified using relatively simple measurements of performance of the executing parallel

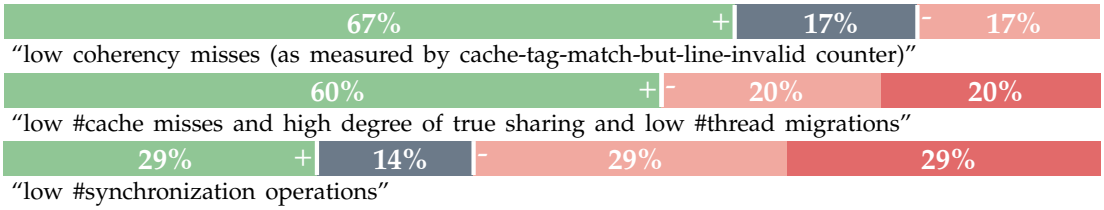

Figure 5.7 – Levels of agreement on observations related to the "**Chains of data dependencies, too little parallelism**" problem.

program.

Indeed, this problem can appear in a great number of different variants that depend on the particular patterns of parallelism, such as pipelined, reduction, or odd-even communication [2]. Ideally we would refine this problem into several sub-problems for the various circumstances in which it can arise. However, we cannot see a clear sub-division of this problem that does not simply degenerate into dozens of special cases. Part of the goal of the expert annotation exercise is to identify these contentious issues where experts disagree.

We believe that finding a better breakdown is an important open (and difficult) problem. One possible starting point is to investigate data dependences within common parallel programming patterns [149, 2]. This would require a higher-level approach to understanding performance problems, based on the parallel orchestration model.

5.4.2 LESS-KNOWN BUT FREQUENT

As we mentioned before, the parallel performance problems in the Figure 5.4 tend to cluster around a rising diagonal line from less-known-and-rare to known-and-frequent. Nonetheless, several problems fall into the other two quadrants. True sharing is on the boundary in terms of familiarity, and the problems of undersubscription and oversubscription are at least somewhat less known but frequent in practice.

True sharing (Figure 5.8). True data sharing occurs when more than one thread accesses data that is updated by at least one thread. Each core that accesses the variable normally has its own local copy of the variable within the core's cache. However, when the variable is updated by one thread, all other copies of the data are invalidated. If a core reads the data again soon, it will find an invalid copy of the data in its cache,

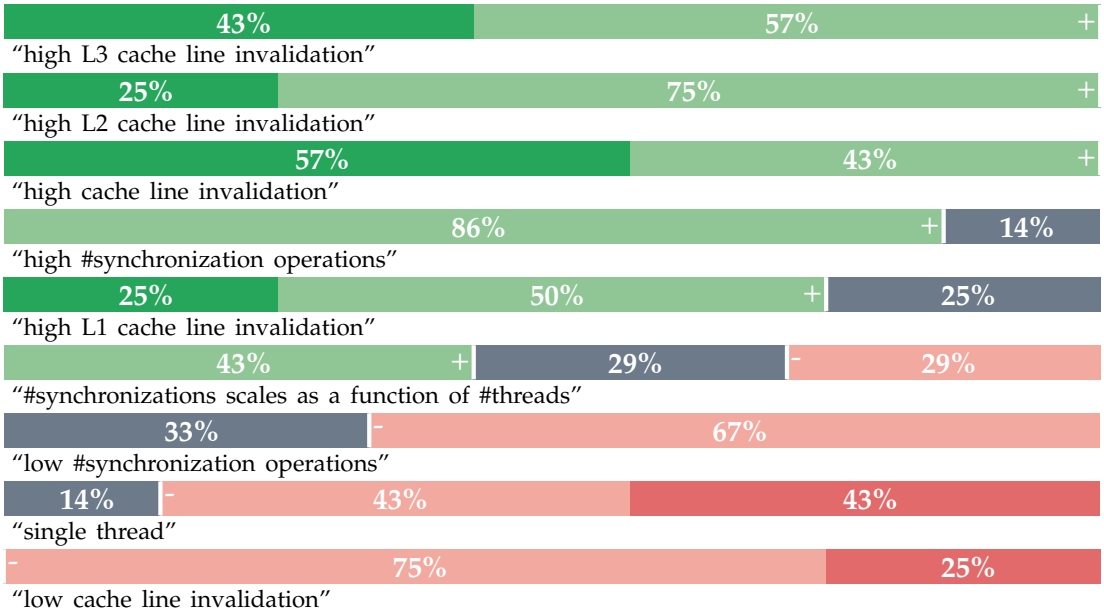

Figure 5.8 – Levels of experts' agreement on observations related to the "**True sharing of updated data**" problem.

and a new copy must be fetched of the updated data. According to our experts, this problem can be detected with the help of hardware counters, which count how many times each core attempts to read data from its cache, and finds that the cache contains a copy, but it is invalid.

High amounts of cache invalidation suggest that true sharing of updated data is sufficiently frequent to cause a performance problem, and a low amount of cache line invalidation suggests the contrary. Note also that the features associated with problematic true sharing can be clearly distinguished from locality performance problems. Cache locality problems are associated with large numbers of cache misses, but cache invalidation misses are associated primarily with (true or false) sharing performance problems.

In other words, true sharing is a problem that arises frequently in practice and experts agree on the diagnosis: it is associated with large numbers of cache line invalidations and large numbers of synchronization operations. However, despite being common in practice and clearly identifiable by experts, 25% of the parallel software developers we surveyed were unfamiliar with this performance problem. This suggests that developers would benefit from information and/or tool support to identify this problem.

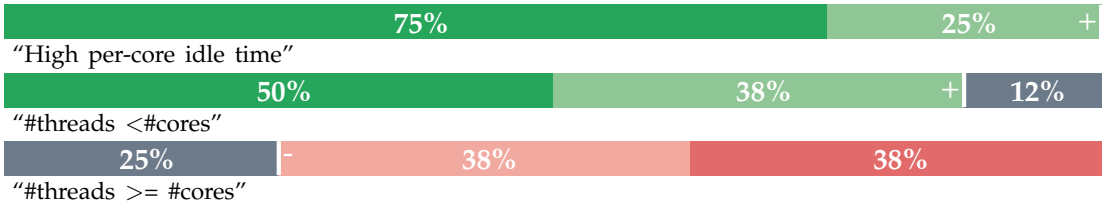

Figure 5.9 – Levels of agreement on observations related to the "**Undersubscription**" problem.

Undersubscription The undersubscription problem (Figure 5.9) is not well known among the parallel software developers we surveyed but still occurs relatively frequently. This performance problem occurs when there are too few active threads for the number of available cores, with the result that the machine is under-utilized. If there is useful parallel work that could be performed by those idle cores, the program could complete that work more quickly if it were to utilize those cores. Our experts are mostly in agreement that high per-core idle time and fewer threads than cores are strong indicators of problematic undersubscription.

Under-utilizing the cores is sometimes deliberate; in some cases parallelism is limited by other overheads, and adding additional threads gives no additional performance benefit. In such circumstances adding more threads may damage performance by introducing more inter-thread communication, more synchronization or more thread management overhead. Using more cores also results in increased power, and if there is no corresponding reduction in execution time the total energy (power $\times$ time) used by the computation will also increase. Nonetheless, our study suggests that in practice it is not uncommon for performance to be constrained by simply not running enough parallel threads.

A very interesting question for future research is why this is so common in practice, given that for many programs it is simple to fix. We speculate that the difficulty is that there is often no simple relationship between the number of executing threads and the speed of the program. One can often fine-tune performance by increasing or decreasing the number of threads. The number of threads may be fixed to a constant in a parallel program to achieve maximum performance on a given target machine. When the software is executed on another machine with more cores, the optimal number of threads may be higher.

Oversubscription The opposite of undersubscription is oversubscription, shown

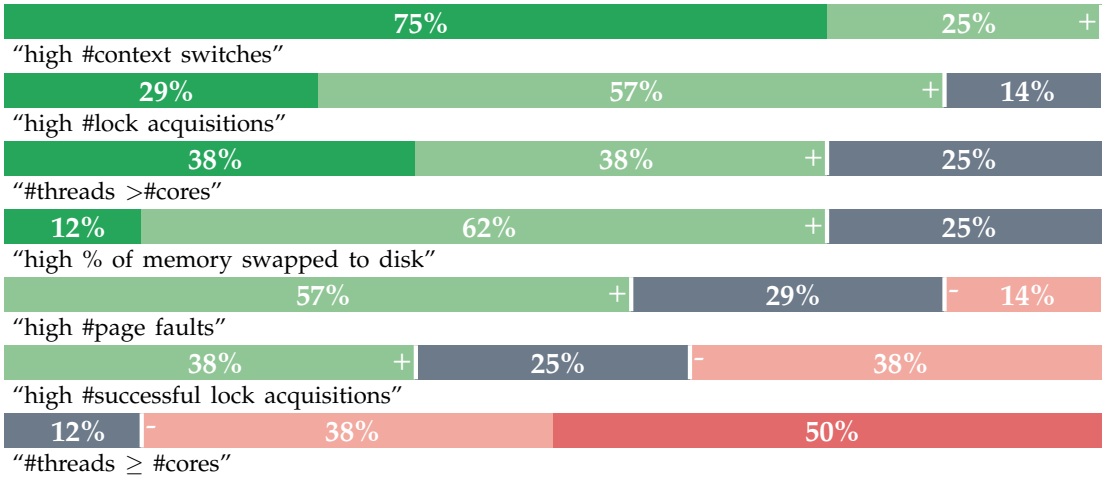

Figure 5.10 – Levels of agreement on observations related to the "**Oversubscription**" problem.

in the Figure 5.10, where the number of threads exceeds the number of cores. Our survey of parallel software developers suggests that problematic oversubscription is at least moderately common. There is a great deal of agreement between our experts that high numbers of context switches, large amounts of synchronization, and more threads than cores are all indicative of problematic oversubscription. Software developers might benefit from guidance or tool support in identifying problematic oversubscription.

The complication with both undersubscription and oversubscription is that they can be beneficial or harmful. There is no simple deterministic relationship between the number of threads and overall execution time. The default strategy used by many is to simply set the number of threads equal to the number of cores. However, naive parallelization strategies can easily result in a great mismatch between threads and cores. For example, a simple parallel divide-and-conquer algorithm might spawn a new thread each time the problem is divided, with the result that the number of threads starts at one and grows with the depth of recursion. Such a program may experience a phase of undersubscription, followed by a second phase of oversubscription. Timeline visualizations of relevant performance metrics might be particularly useful in such cases.

Although it can be remarkably difficult to judge the best number of threads to get maximum parallel speedup, the results of our survey are clear on some points. Both under- and oversubscription are moderately common in real programs, but not well

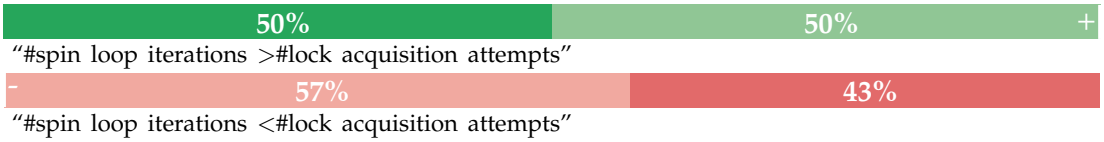

Figure 5.11 – Levels of agreement on observations related to the "**Badly-behaved spin-locks**" problem.

recognized by developers as potential performance problems. Fortunately, there is agreement among our experts about which metrics might indicate problematic under- or oversubscription. This suggests that tools and/or information may help developers to identify when under/oversubscription is problematic.

5.4.3 LESS-KNOWN AND INFREQUENT

The less-known and infrequent parallel performance problems of the Figure 5.4 tend to be rather technical and obscure. As mentioned at the start of section 7.4 many of the problems in this quadrant are related to software interactions with hardware or the operating system at a low level. For example, to understand false sharing one must understand how parallel computers maintain coherency between copies of the same data in different caches during updates.

It is interesting that these are regarded as rare among those developers who are familiar with them. This leads to the question that if these problems are indeed rare, can we simply ignore them? The difficulty with ignoring these problems is that even if they are rare, some can have a catastrophic impact on performance. For example, badly-behaved spinlocks can bring a parallel application almost to a halt. Several threads repeatedly updating different, but adjacent, array locations can cause large slowdowns due to false sharing. Even if these problems are rare, their large impact means that they cannot simply be ignored.

If we consider badly-behaved spinlocks in more detail, we see a great deal of agreement between our experts on how the problem might be diagnosed (see Figure 5.11). There is also a much agreement (although not unanimity) about the observations relating to false data sharing (Figure 5.12). This is in agreement with the literature, where cache invalidation is clearly identified as a key symptom for true/false sharing [**?**]. Similarly, the experts broadly agree on the observations that might indicate a prob-

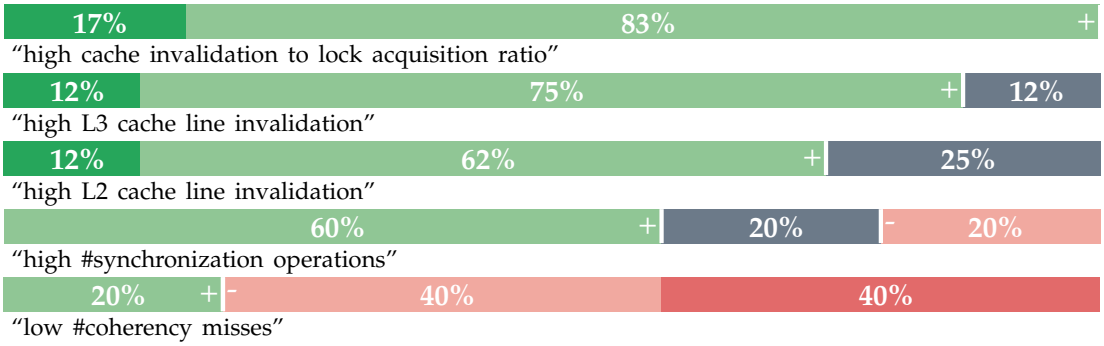

Figure 5.12 – Levels of agreement on observations related to the "**False data sharing**" problem.

lem with translation look-aside buffer (TLB) locality (Figure 5.13), which is another problem that can have a large impact on execution time.

It appears that these problems are unfamiliar to many parallel software developers because they are related to quite low-level interactions with the parallel computer architecture or operating system. However, the fact that they relate to low level technical features gives a significant degree of hope that they can be diagnosed with low-level hardware and operating system performance counters. Our experts are largely in agreement about the observations associated with these problems. This suggests that these problems are good candidates for tool support that specifically searches for these problems among performance data. The problems may continue to be rare and unfamiliar to many parallel software developers. But on the rare occasions that these problems arise, tools may be able to detect at least the possibility of their existence, and communicate the problem and possible solutions to the developer.

5.4.4 THREATS TO VALIDITY

As both the survey and validation study are based on understanding of hardware issues which are potentially very complex, there is a possibility of misunderstanding and misdiagnosis on the part of the participants. While this initial exploration deliberately targeted a broad class of programmers, the context in which both experts and programmers work also introduces a potential confound. The disagreement among experts regarding some observations points to a need for further investigation of these issues in a context-specific way. We would recommend that both the architecture, the

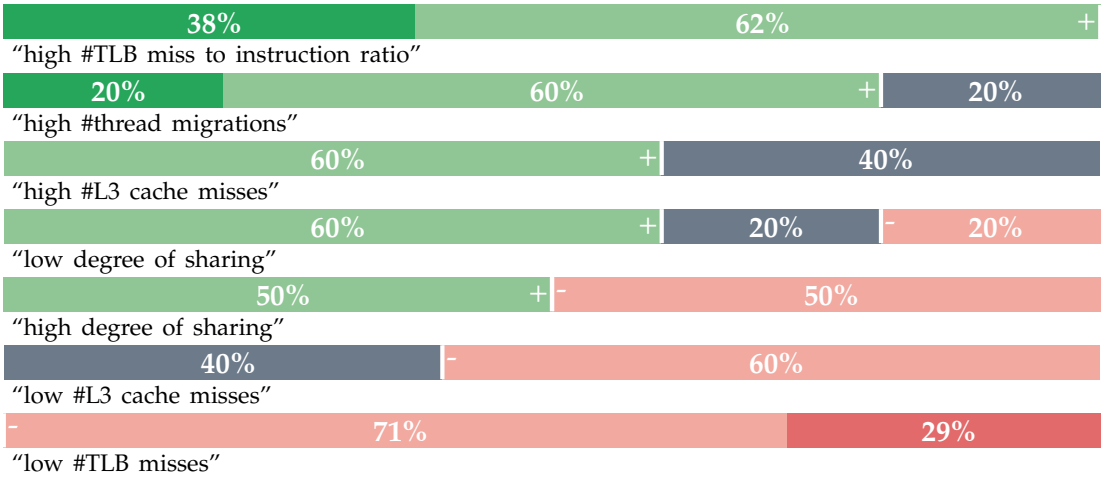

Figure 5.13 – Levels of agreement on observations related to the "**TLB Locality**" problem.

organizational context, the programming language/environment, and the type of software being produced are considered within future work.

Nonetheless there are areas of significant agreement among the participants, in areas such as lock contention and locality. One might conclude that these are areas so well recognized and understood that the high level of agreement is an inevitable outcome of the study. However, we believe that there is value in distinguishing areas of consensus from areas of doubt.

5.4.5 MODEL APPLICATIONS

The model and taxonomy provide a description of the problem space for the diagnosis of parallel performance problems. The model is intended to provide a foundation for the development of tools and hence is based on data that we can measure or derive; it relates observations from concrete instrumentation data to broad categories of problems.

At the highest level, having a list of potential performance problems makes it easier for tool designers to discuss the different situations that a programmer might face, and the observational component of the model provides a starting point for discussing the information that they might find useful in diagnosing their own program. Different forms of tool support might be envisaged:

- Interactive performance debugging tools can be created, visualizing data asso-

ciated with various indications or contra-indications and highlighting relevant data to users. Groupings of particular problems from the taxonomy might be addressed within the same tool or same part of a tool. For example, we could develop a dashboard visualization for data locality which makes available the most relevant information for the diagnosis of several different data locality problems.

- Automated tools can be created for more efficient parallel performance diagnosis and prediction. Ideally, a range of problems would be recognized by the same tool or an integrated suite of tools. Such automated support could also be integrated into visualization tools as described above.

- Systems can be intelligently instrumented to give strong indications of the presence of various performance problems and automatic watches/alerts can be fired if a particular observation exceeds a user-defined threshold.

There are a number of potential advantages to a tool that focuses on a taxonomy of specific parallel performance problems, rather than performance data alone. Focusing on a set of potential problems gives the developer structure in their efforts to improve parallel performance. Indeed, prior research on large-scale parallel computing systems has focused on identifying problematic scalability, communication and message-passing problems [76, 47]. It may also introduce developers to performance problems that were previously unknown to them. Finally, a tool that deals with specific problems may be able to direct the developer towards possible solutions. For example, once a programmer has discovered a problem with false data sharing, a tool can provide information on solutions to common causes of the problem. Helping developers fix parallel performance problems is an important area of future research.

For performance visualization systems in particular, the model provides suggestions on the type of performance metric that should be collected, and describes how this information can be related back to various performance problems that the developer might need to diagnose. For problems where there are distinct phases of program execution, timeline visualizations are likely to be useful, but we have also seen in section 5.4.2 that the same algorithm might also exhibit different performance behaviour over time (e.g. moving from undersubscription to oversubscription within a parallel divide and conquer algorithm). Ultimately, the model is designed to support the de-

veloper, who is in the position to discriminate between problems and assess whether a particular value for an observation is "high" or "low" in the context of their own development project.

Going up a level from the data locality example above, consider a performance visualisation tool that aims to provide the developer with insight into whether a memory-related problem exists in a multi-threaded program. To design such a tool, we first need to know what kind of memory problems programmers might be faced with, the most common being cache locality (Figure 5.5), chains of data dependencies and true sharing. Given the observations that are strong indications or contra-indications, we should include in our tool some visual representation of relevant measures such as cache misses and cache invalidations. While we might not know how to visually encode high and low values for the counters, we might consider making such counts relative to the total execution time (and hence providing an estimation of performance impact), therefore having relative measures that are more easily understood.

The model and taxonomy may also be useful for those involved in teaching parallel programming, conducting software engineering research on the practice of parallel programming, and identifying gaps in our knowledge of, and ability to diagnose certain problems.

5.5 CONCLUDING REMARKS

The switch from single core to multicore architectures has created new challenges for software engineering. Whereas parallel programming was once confined primarily to the supercomputing community, we now find multicore processors in desktop, laptop and even embedded systems. Typical software developers now face the challenge of building parallel software for a wide variety of applications. This creates a need for new tools, education and software development methods.

One step towards solving these problems is to identify the kinds of performance problems that developers encounter when building software for multi-core machines, and how those problems might be diagnosed. We have presented here one such taxonomy, grouped into a number of broad, interrelated themes. Our model focuses primarily on concrete problems that have potential to be related to easily-collectable

data, rather than more abstract problems relating to the software architecture or overall design (although this is also an important aspect).

This model has been validated with experts, identifying areas of high agreement, which, when combined with data on relative frequency of occurrence, provides promising directions for initial tool support. Our results indicate that there is significant agreement among developers and experts about many of the parallel performance problems identified, and in particular about how the problems might be diagnosed. Our study has also identified some contentious issues. Resolving these areas of disagreement might not involve finding the "right" answer but rather a more nuanced analysis of the problem. The observation might be context-dependent or require simultaneous consideration of multiple pieces of data.

As well as being useful to tool builders, the taxonomy might also provide a useful starting point for educators as students are often at a loss to understand parallel performance of real programs, partly because they are unaware of the kinds of problems that might exist.

Finally, we hope that our taxonomy will be a useful starting point for future research on understanding and diagnosing parallel performance problems. We expect that future research will further refine our taxonomy, or propose entirely new ones. Differences in how programmers with different levels of expertise diagnose parallel performance problems is also deserving of further investigation.

CHAPTER 6 ANALYSING DATA LOCALITY

The previous chapter explored the problem space for parallel software performance optimisation. In order to progress towards the exploration of the solution space, while building a general foundation, we look at the issue of *Data Locality* identified in our taxonomy in the previous chapter, which is arguably one of the most important challenges currently faced by programmers of parallel systems.

In this chapter we deepen our analysis of the problem, based on the observational model. We also explain the process we went through in order to collect and process the necessary information to build a visualisation tool for data locality problem diagnosis.

6.1 CPU, MEMORY AND CACHES

Prior to diving into the diagnosis of data locality problem which we have described previously in our taxonomy, we must clarify some of the general concepts with regard to CPU architecture. This is necessary, as to understand the problem we must comprehend the context and technical causes which lead to data locality problems.

Modern systems, typically have several CPUs which represent resources shared by all running software with the help of the kernel scheduler of the operating system. Moreover, modern processors also provide multi-level hardware caches for improving I/O performance, such caches are located on the chip itself and become smaller and faster the closer they are to the CPU itself. For example, the *Intel Xeon E7-8870 v3* processor consists of 16 distinct cores, each core has a closely-located Level 1 cache of 32 KB and a bigger, but slower Level 2 cache of 256 KB. Furthermore, it also provides a Level 3 cache of 45 MB, shared between all cores. To illustrate this, a generic dual-core processor architecture is depicted in the Figure 6.1 where each core has its own Level 1 and 2 caches and the last level cache is shared between both cores present on the chip.

A common computer system typically has more threads and processes than available CPUs and multiple threads might need to run simultaneously. To accomplish this, the operating system maintains a queue of software threads (*ready to run queue*) and schedules the execution when the processor becomes available. In multi-processor systems, the kernel usually tries to keep software threads in the same run queue, where their caches are located. These caches are usually described as having *cache warmth* while the approach to favour a particular CPU is known as *CPU affinity*.

Once scheduled on a CPU, each thread is allowed to execute some instructions, which are chosen from the CPU instruction set. An instruction including several steps which are processed by a component of the CPU known as *functional unit*, such as *arithmetic and logic unit* or *memory address register*. Each instruction consists of several steps:

1. **Instruction Fetch.** The instruction is fetched from the memory address specified in the program counter (PC) register, then stored in the instruction register (IR).

2. **Instruction Decode.** The encoded instruction present in the IR is interpreted by the decoder. The decoding process allows the CPU to determine what instruction is to be performed, so that the CPU can tell how many operands it needs to fetch in order to perform the instruction.

3. **Execute.** The *control unit* of the CPU passes the instruction to the relevant *function unit*. The result generated by the operation is stored in the main memory, or sent to an output device.

4. (Optional) **Memory Access.** In case of an indirect memory instruction, the effective address is read from main memory and any required data is fetched to the appropriate registers.

5. (Optional) **Register Write-Back.** The result of the operation is written back to a register.

Each of these steps takes at least one clock cycle in order to be processed: memory access being the slowest and may take dozens of cycles to read or write to main memory, during which instruction execution is *stalled* and those cycles are known as *stalled cycles*. This is the main reason why CPU caches exist to begin with, as they allow us to reduce the amount of cycles needed for memory access.

Another component of CPU architecture is known as *instruction pipeline*, allowing each CPU to execute several instructions in parallel by executing different steps of different instructions at the same time. This is analogous to a factory assembly line, where stages of production can be executed in parallel, thus increasing overall throughput. As on the list above, a processor can process a fetch, decode, execute and access memory simultaneously with the goal to execute an instruction per cycle.

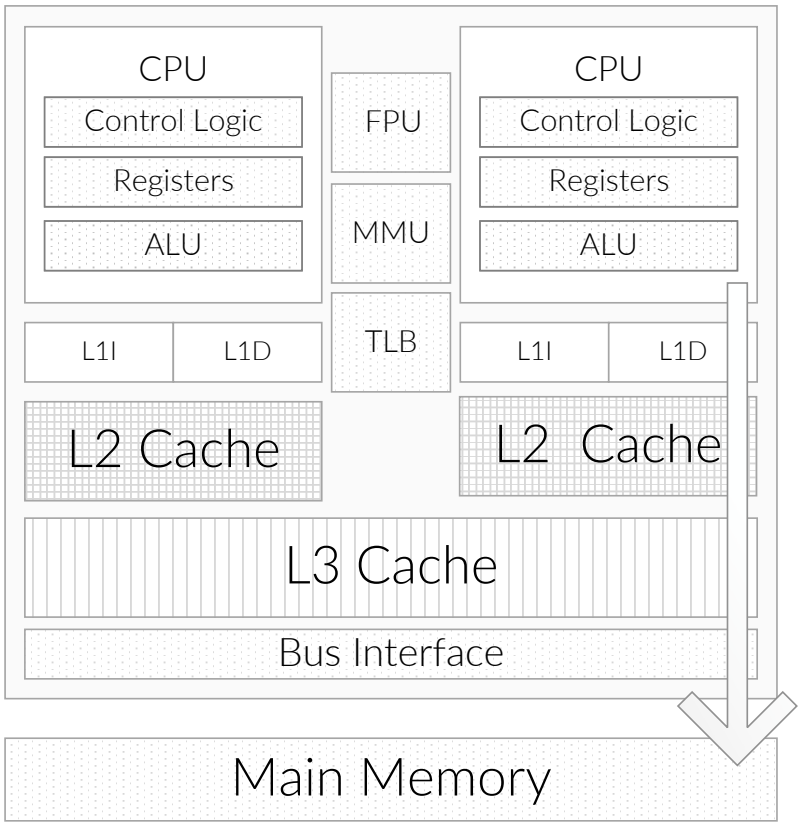

Figure 6.1 – Conceptual representation of a generic dual-core processor.

Data locality problems occur when during the memory access, the data is not present in a reasonably nearby location, resulting in more distant cache or main memory fetches. In turn, this causes pipeline stalls as other instructions can not be executed until the the memory access completes. This process varies depending on the architecture and this is simply to illustrate the cause of the data locality problems.

There are two main types of data locality: **temporal** and **spacial** locality of reference. Good temporal locality is achieved by the reuse of specific data, and/or resources, within a relatively small time duration. Good spatial locality is achieved by the use of data elements within relatively close storage locations, such as L1 or L2

Event	Latency	Scaled
CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access	120 ns	6 min
Solid-state disk I/O	50-150 μs	2-6 days
Spinning disk I/O	1-10 ms	1-12 months
Internet: SF to NYC	40 ms	4 years
Internet: SF to Australia	81 ms	8 years
Physical System Reboot	5 min	32 millenia

Table 6.1 – Example Time Scale of System Latencies [59]

caches.

To balance between size and latency, multiple levels of cache are used. The access time for Level 1 cache is usually only a few CPU clock cycles and for larger Level 2 cache, the access time is around a dozen clock cycles. Main memory access can take around 120 nanoseconds, which is about 360 cycles on a 3 GHz processor.

In the Table 6.1 one can see how big the differences in latency are. To demonstrate the differences in time scales, the table shows an average time each event might take when scaled to an imaginary system where a CPU cycle takes one second to execute. The table also shows disk and network accesses, as they are a natural extension of data locality but not addressed specifically in our analysis.

In the Figure 6.1 between the cores one can find the *memory management unit* (MMU) along with *translation look-aside buffer*. The MMU is responsible for virtual-to-physical address translation and uses a TLB to cache address translations when a Level 1 cache miss occurs. This address translation process takes time and the TLB capacity remains relatively small, usually under a thousand entries. Similar to caches, TLBs may have multiple levels, for example a small and very fast Level 1 TLB along with a larger Level 2 TLB that is somewhat slower.

Finally, modern servers typically have multiple processors and using non-uniform memory access architecture (NUMA). In such architectures, the processors are interconnected and the memory access depends on the location of the processor, as demonstrated in the Figure 6.2 where if a hypothetical *CPU 1* needs to access the data located in *DRAM 4*, the data needs to be transferred through the memory bus between *CPU*

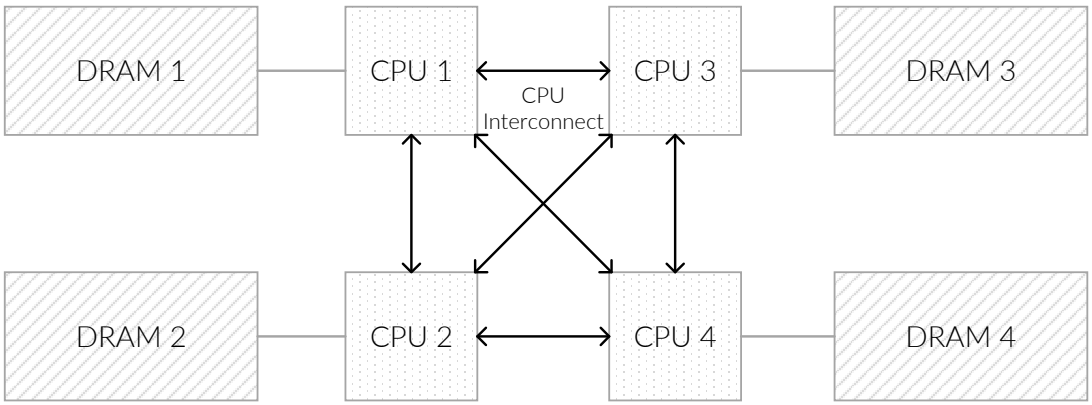

Figure 6.2 – Conceptual representation of the NUMA interconnect architecture.

4 and *DRAM* and then through the interconnect bus between *CPU 1* and *CPU 4*. This requires some memory accesses to go through another CPU in order to perform memory read/write operations. This further increases latency and may even affect the performance of other threads running concurrently on other CPUs.

6.2 APPLYING THE OBSERVATIONAL MODEL

We have previously identified five performance problems within *Data Locality* category, more specifically **Poor Cache Locality**, **Poor TLB Locality**, **Unnecessary DRAM Memory Paging**, **NUMA memory shared between CPUs** and **Page faults**. In this section, we are going to apply the observational model to identify the necessary information to be presented within our performance analysis tool.

Due to the way modern processors are designed, they access memory from various levels of caches in order; this makes each subsequent memory fetch slower as larger and more distant memory banks are accessed. A simple conceptual way to view this is illustrated below and in the Figure 6.1 with the arrow on the left of the diagram, going through various components in the chip.

1. Processor attempts to retrieve the data from L1 cache.
2. If not found previously, the virtual-to-physical address translation is performed along with an attempt to retrieve data from L2 cache.
3. If not found previously, processor attempts to retrieve data from L3 cache.

4. If not found previously, processor attempts to retrieve data from RAM, with a potential involvement of the interconnect on multi-processor systems.

5. If not found previously, a page fault is raised and the appropriate memory page is loaded into main memory.

Now that we have gone through the components and the process flow involved in the data locality problems, we need to elicit the measurable events and counters required to effectively diagnose the problem. Drawing on the model we have presented in Chapter 5, we need to look at and pick relevant observations, ideally with high agreement levels and discarding highly controversial observations (i.e. observations with a high number of "irrelevant" annotation by experts and observations with contradictory annotations).

1. **Cache Locality.** All of the observations seem to have general agreement between experts on whether an observation is a indication or contraindication of this problem, with "high #L2 cache misses" being the most agreed indication and "low cache misses as measured with hardware counters" being the strongest contraindication.

2. **TLB Locality.** The observations *"high #TLB miss to instruction ratio"* and *"low #TLB misses"* have the most discriminatory potential with all experts agreeing on them being respectively an indication and a contraindication of the TLB locality problem.

3. **Sharing of data between CPUs on NUMA systems.** The "high remote memory access" observation is the only observation that seems to have an inter-rater consensus.

4. **Unnecessary DRAM Memory Paging.** All of the observations have high levels of agreement among experts, with the simple count of DRAM page changes having high discriminatory power, as a high number of DRAM page changes being an indication and a low number of DRAM page changes a contraindication of a problem.

5. **Page faults.** This particular problem seem to have only a single inter-rater consensus with all experts annotating "high #page faults" as an indication of the

problem and relatively strong agreement on "very low #page faults" which is considered as a contraindication.

The usage of the model is rather straightforward, as once we have identified the required observations, we need to identify the exact counters, events or metrics we require in order to successfully diagnose each problem. Below we enumerate such raw items, along with its collection mechanism. This is only part of the solution, as the information needs to be further refined in order to be useful for programmers.

1. **Memory reads and writes.** Modern hardware contains a counter that can allow us to measure the amount of memory reads and writes for a particular processor, in bytes of memory.

2. **L1, L2 and L3 cache misses.** Similarly, there are counters in hardware that would allow us to measure cache misses on a particular level. This event essentially occurs when data has not been found on a particular level and the processor needs to fetch it from a higher (and slower) level.

3. **L1, L2 and L3 cache hits.** We can also obtain values representing the number of successful hits in various levels of cache, representing successfully obtained data.

4. **TLB misses.** Once again, we can retrieve the number of TLB misses from the hardware performance counters on most of the modern CPUs.

5. **Minor Page Faults.** Operating system events can be used to get information about minor and major page faults. A minor page fault represents a memory address that is loaded in memory, but is not mapped to a memory management unit, resulting in the system merely making an entry for the particular address.

6. **Major Page Faults.** Operating system events that are raised by hardware when a program attempts to access memory that is mapped in the virtual address space, but not loaded in hardware. This often requires the operating system to swap or load from disk, resulting in a considerable slowdown.

7. **Processor Interconnect Traffic.** In most modern CPU architectures, hardware performance counters can be used to extract information about the memory traf-

fic passing through the interconnect bus, such as Intel QuickPath Interconnect (QPI) or AMD HyperTransport (HT).

6.3 DIAGNOSIS PROCESS

The observational model allowed us to identify the information that should be presented to the end-users (programmers) within our tool, assuming that if programmers are able to make specific observations, such as seeing a *high number of L2 cache misses*, they will be able to identify potential problems.

Figure 6.3 presents a decision tree describing a possible diagnosis path for identifying problems with poor data locality. Such a decision tree provides a simple representation of how the different pieces of information within a visualisation might be used in the process. This would fit under a more broad framework, as the decision tree addresses solely data locality issues, so it can be seen as a sub-tree of some larger decision tree. To create the tree, we have selected the most discriminatory observations from the observational model.

Although we have already identified various components of computer architecture that can be responsible for slow memory access, from the programmers' standpoint the cause of poorly performing programs is usually *random memory access patterns*. In other words, accessing memory in a non-sequential way is generally harmful to the performance of cache and memory as caches work on the expectation that the data that has recently been touched and the data that is located closely in memory to it is more likely to be accessed again.

Random access patterns generally contain limited memory reuse, leading to more cache misses as the probability of finding the required data in the cache is low. Moreover, it also leads to lower utilisation of a cache line. Data is transferred between memory and cache in blocks of fixed size, usually 64 bytes which are called *cache lines*. For example, if a single 32-bit integer (4 bytes) variable is accessed within a program, an entire block of 64 bytes of surrounding memory is brought into the cache.

Modern processors also integrate a hardware *prefetcher*, which relies on finding regular access patterns to determine what data to pre-load, and thereby hide memory access latencies. Random access patterns therefore make the hardware prefetcher in-

effective or may even trick the hardware prefetcher into prefetching data that is not useful, wasting memory bandwidth.

Random access patterns can originate from different sources, including *data structures*, *dynamic memory allocation* and *algorithms*. Some data structures inherently cause random access patterns, for example, most implementations of traversal of tree data structures or hash tables. Programmers can greatly improve performance by replacing such data structures with more cache-friendly implementations.

Changing random access patterns is generally hard: sometimes it might require an alternative algorithm with better cache behaviour, however, such algorithms can be inferior in other ways; other times it may be possible to adapt data structure to the algorithm. For example, it may be possible to sort an adjacency matrix so that nodes that are connected are close to each other, so that following the edges results in a more regular access pattern.

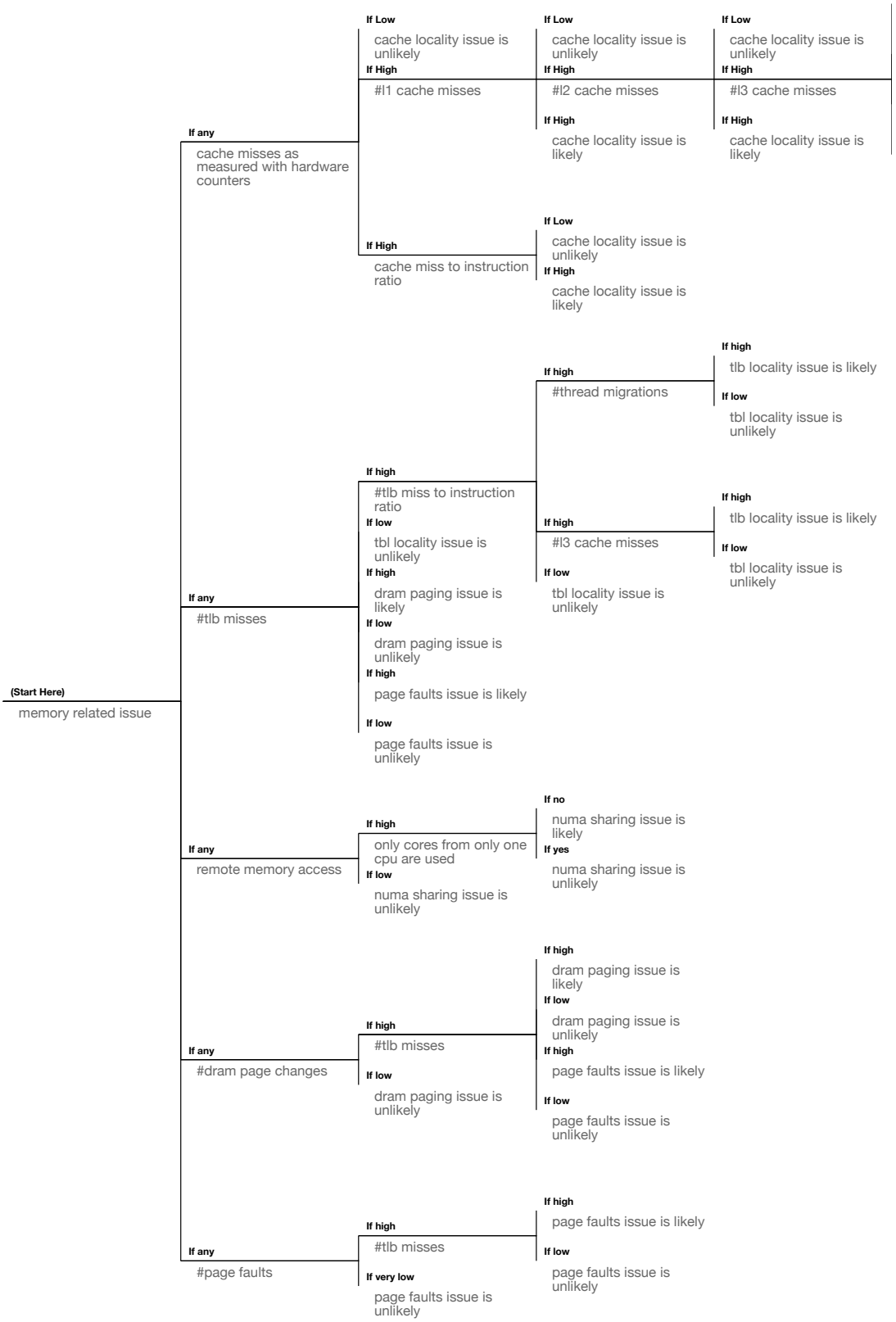

Figure 6.3 – Data locality problem diagnosis tree.

6.4 DATA COLLECTION

As we described in the previous section, we were able to understand which data types are necessary to construct a picture representing memory traffic. However, our next natural step was to understand technically, how we are going to collect the data and which specific features we need to measure.

As our interview results suggest that we need to support orchestration models and scoping, we therefore concluded that every measurable event needs to have the following dimensions:

1. **Processor.** The specific processor (core) that raised the event.
2. **Process.** The specific process that raised the event.
3. **Thread.** The specific thread that raised the event.
4. **Call Stack.** The instruction pointer or a call stack that can be used to determine which function raised the event. Processor, process and thread information allows us to map each data point to a particular semantic worker. It presents fine-grained, verbose information, it also allows us to re-sample data in different ways, summing and averaging events per process, processor or even per machine.

Cache misses, cache hits, memory reads and writes are architecture specific events. In order to access and collect the data, we need to sample the values from hardware performance counters. However, as the name suggests, hardware performance counters cannot be associated with a specific process or a specific software thread, as it only gives hardware information. This is problematic as we determined that we need processor, process and thread information for each event. Therefore, we must up-sample the hardware counters in order to enrich them with the required information. The up-sampling, or increasing the sampling rate of a signal, can be performed if we bring in detailed context switch information.

A context switch represents a process of storing and restoring a state of a process or thread so that execution can be resumed from the same point in time later. Since hardware counters provide us with cache misses, cache hits and memory usage per

processor, which means we can up-sample only if we know which thread was executing at a given point in time. In modern operating systems such as Windows, the operating system provides a facility that allows us to collect the context switch information. This facility is called Event Tracing for Windows and was originally introduced in Windows 2000. It provides application programmers the ability to start and stop event tracing sessions and record (consume) specific events. By knowing precisely which thread executes at which point in time, we can up-sample and approximate hardware performance counters to measure the counters per thread, adding valuable information.

6.5 MEASURING THE PERFORMANCE IMPACT

In order to effectively evaluate our tools, we needed to use a benchmark suite that illustrates a data locality problem, while being relatively easy to understand and optimise.

We have based our initial analysis, tests and benchmarking on an algorithm commonly used in parallel programming optimisation: a matrix multiplication algorithm. As a quick reminder, the definition of matrix multiplication is that if $C = AB$ for an $n \times m$ matrix A and an $m \times p$ matrix B, then C is an $n \times p$ matrix with entries:

$$C_{ij} = \sum_{k=1}^{m} A_{ik} B_{kj}$$

In this section we will analyse a matrix multiplication algorithm and we will attempt to understand what the performance problems are and how a conclusion can be reached as to what is causing the problem.

6.5.1 PARALLEL IMPLEMENTATIONS

Figure 6.4 is a C++ function for matrix multiplication, parallelised with an outer parallel for loop, a common construct that can be found in many parallel programming libraries such as Microsoft Parallel Programming Library or Intel Threading Building Blocks

```
void MultiplyMatrix(int n, int** a, int** b, int** c)
{
    parallel_for (0, n, [&](int k)
    {
        for (int j=0; j<n; j++) {
            int r = b[k][j];
            for (int i=0; i<n; i++)
                c[i][j] += a[i][k] * r;
        }
    });
}
```

Figure 6.4 – Function performing matrix multiplication, parallelized with an outer parallel **for loop**, a common construct that can be found in many parallel programming libraries such as Microsoft Parallel Programming Library or Intel Threading Building Blocks.

When running the above program on a commodity quad core machine, and multiplying two 1000 by 1000 matrices together, the total execution time we measured on our test machine is around 4 seconds; this is already 4 times faster than its sequential sibling. We know that the multiplication algorithm results in $1000 * 1000 * 1000$ (one billion) sub-multiplications. Each sub-multiplication needs to read 2 addresses and writes into another one, resulting in 2 billion memory reads in total.

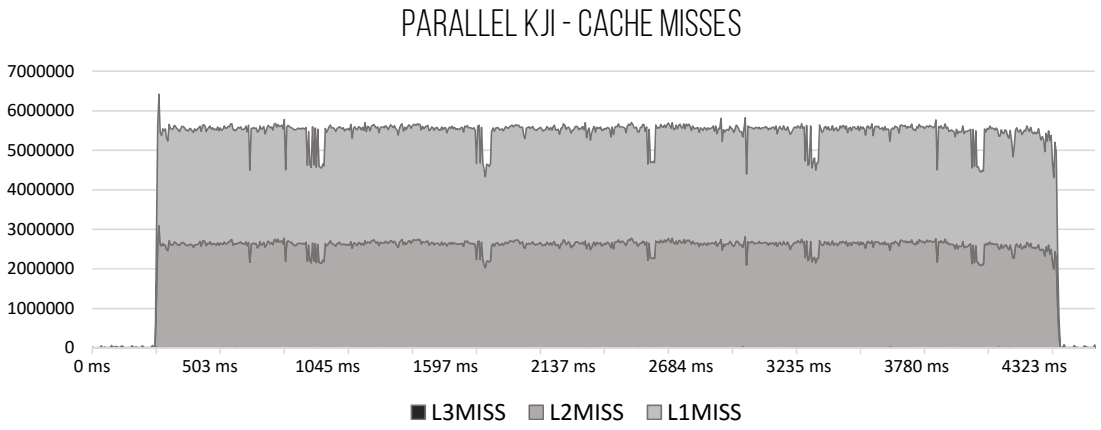

Figure 6.5 – Data representations of cache misses over time that occurred due to an experimental run of a parallel matrix multiplication program.

However, this performance might be insufficient and can be improved. By running our data collection utility, we can extract the number of cache misses and various useful metrics per thread which represent how effectively the cache is being used. In the Figure 6.5, one can see that a significant amount of L1 and L2 cache misses occur, while the amount of L3 misses remains insignificant in comparison. From this, we can

conclude that the current implementation stores the entire work set (both matrices) in L3 cache, but accesses them in an inefficient manner.

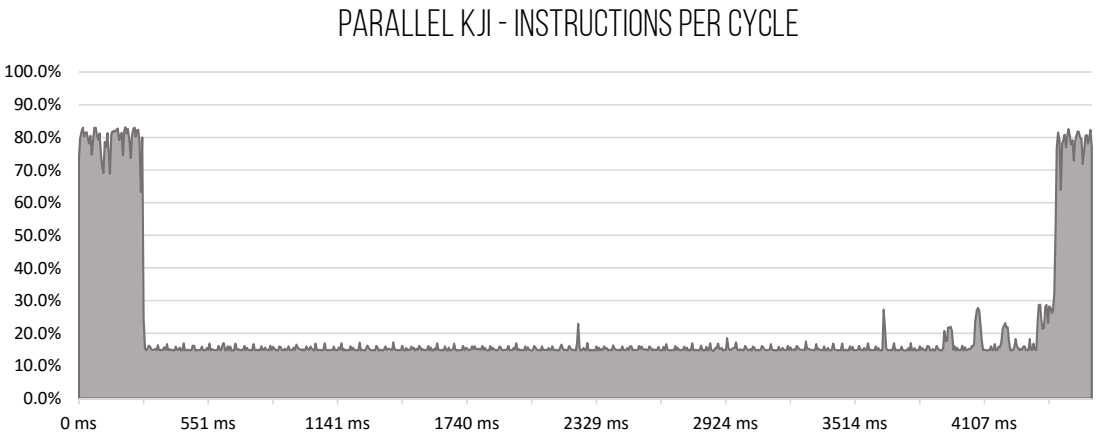

Figure 6.6 – The average number of instructions executed for each clock cycle.

There are several other metrics which can be useful for our diagnosis, in addition to raw L1, L2 and L3 cache miss counts. One of them can be seen in the Figure 6.6 and shows the average number of instructions executed for each clock cycle. A significant decrease in executed instructions can be observed during the execution of our matrix multiplication program. This is due to the processor not being able to effectively execute instructions and the processor pipeline stalling and waiting for data to arrive from the cache.

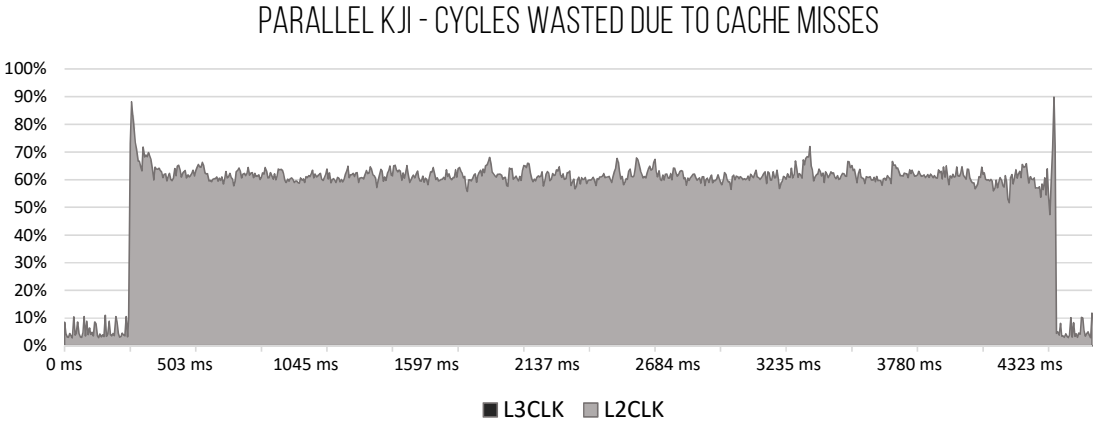

Figure 6.7 – The clock cycles wasted due to L2 and L3 cache misses.

As we can see in the Figure 6.6, the instruction decrease demonstrates clearly the slowdown, since the CPU is not executing instructions and instead is waiting for data to arrive to the pipeline. So far, we have formed this hypothesis by simply looking at

```
void MultiplyMatrix(int n, int** a, int** b, int** c)
{
    parallel_for (0, n, [&](int k)
    {
        for (int i=0; i<n; i++) {
            int r = a[i][k];
            for (int j=0; j<n; j++)
                c[i][j] += r * b[k][j];
        }
    });
}
```

Figure 6.8 – Function performing matrix multiplication, parallelized with an outer parallel **for loop**, a common construct that can be found in many parallel programming libraries such as Microsoft Parallel Programming Library or Intel Threading Building Blocks.

two particular graphs, however the "performance impact" is not clear and is difficult to quantify.

Figure 6.7 shows a different perspective of looking on the "performance impact" of data locality. Here, we have performed an estimation of the cycles wasted due to L2 and L3 cache misses, more details are provided on the actual calculation in the following section. This particular graph allows us to combine both previous graphs in a single one and allows us to identify the cache level that becomes a bottleneck in our program, Cache L2 in this particular case.

There are different ways to implement the matrix multiplication algorithm and a more effective implementation can be obtained by simply swapping the order of accesses. The algorithm illustrated in the Figure 6.8, takes less than a second to execute on the same quad core machine — this is almost a 16 times speedup compared to its worst-case sequential sibling. The efficiency of this algorithm comes from the data access pattern — the elements of the matrices being multiplied are accessed sequentially as laid out in memory.

Data locality access patterns matter. In our matrix multiplication benchmark suite, by simply swapping for loops and changing how memory is accessed, the number of cache misses can be significantly reduced (Figure 6.9), which means that the processor spends less time loading the data and more time actually doing the calculation.

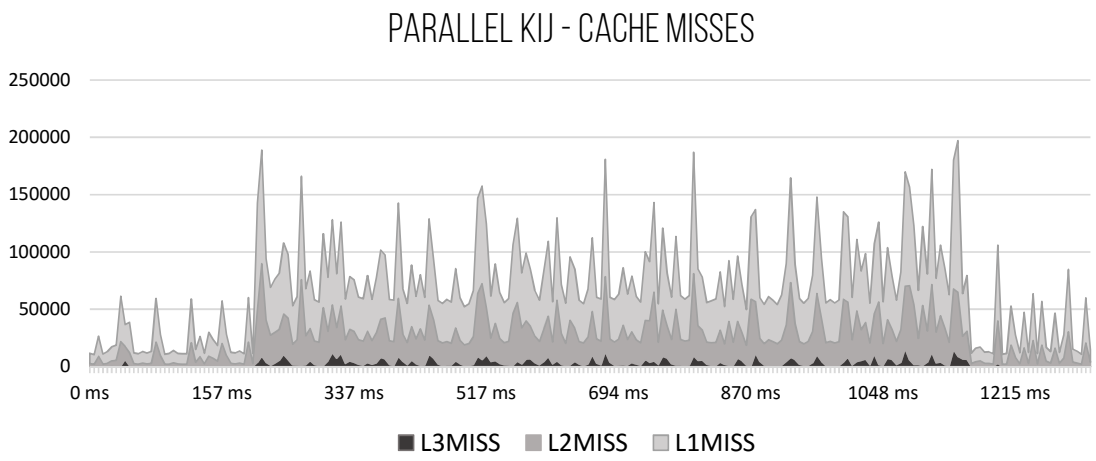

Figure 6.9 – Data representations of cache misses over time that occurred due an experimental run of a parallel matrix multiplication program.

6.5.2 LOST CYCLES

As we have seen in the Figure 6.7, expressing various levels of cache in terms of cycles can be a good estimator of performance impact; but how do we calculate and extend this to cover other data locality components?

Luckily, Intel provides a good starting point with some metrics used in their *Intel Performance Counter Monitor (PCM)* and *Intel VTunes* tools which allow estimation of the performance impact with more granularity. One such metric is shown below and can be used to calculate the performance impact of L3 cache misses in terms of CPU cycles.

let $x_\Delta = x_t - x_{t-1}$ *for any* x:

$$L1Impact = \frac{10 * L2Hit_\Delta}{CpuClkUnhaltedThread_\Delta}$$

For example, as shown in the formula above, we can calculate the performance impact of L1 cache misses simply by counting the number of L2 hits and dividing this by the number of elapsed cycles. As we are interested in calculating the impact of L1 cache misses, we are not interested in the L1 cache misses that also miss L2. We also consider here that a "cost" of an L2 miss is roughly 10 CPU cycles.

$CpuClkUnhaltedThread$ represents the number of core clock cycles on a specific core that the thread is running; this counts only used cycles, as the halted cycles are not

counted. To clarify this, all x86 CPUs have an instruction known as `HLT` ("Halt"), which puts the CPU into an idle state, and the CPU can be brought back to life if it receives an interruption.

let $x_\Delta = x_t - x_{t-1}$ *for any* x:

$$L3Impact = \frac{180 * L3Misses_\Delta}{CpuClkUnhaltedThread_\Delta}$$

Similarly, *L3Impact* is a ratio which represents how many core cycles were potentially lost due to L3 cache misses. Level 3 cache misses are calculated using a hardware performance counter *L3Misses* and the difference is taken between two counter states ($x_t - x_{t-1}$).

We can also estimate the performance impact of Level 2 cache misses, by estimating how many core cycles were potentially lost due to missing the L2 cache but still hitting the L3 cache. This is significantly more difficult, as the Level 2 cache is an intermediary level cache. The formula below results in a ratio which is usually between 0 and 1. In some cases, however, this ratio could be > 1.0 due to a lower estimation of access latency, according to Intel documentation.

let $x_\Delta = x_t - x_{t-1}$ *for any* x:

$$L2Impact = \frac{35 * L3UnsharedHit_\Delta + 74 * L2HitM_\Delta}{CpuClkUnhaltedThread_\Delta}$$

In this metric, *L3UnsharedHit* refers to the number of retired loads that hit valid versions in the last level cache, *L2HitM* is the number of memory load instructions retired that hit modified data in the sibling core, due to false or true data sharing.[1]

In other words, this metric attempts to approximate the performance impact in terms of cycles by assigning different weights (35 and 74) to a pair of hardware performance counters and adding them together. Noticeably, Level 2 hits that hit the modified data are weighted more than twice the the actual L2 misses that hit L3.

To illustrate this, consider a scenario where an array of four 32-bit integer numbers $\{A|A \in \mathbb{Z}\} = \{1,2,3,4\}$ (A is very small and fits in a single cache line) is accessed by two different threads scheduled on CPU cores: C_1 and C_2. Initially, both cores have the

[1] True and False data sharing are explained in Chapter 5.

entire A in their respective Level 2 caches. If C_1 has modified the first two elements of A ($\{1,2\}$) and then C_2 attempts to read any of the elements of A, this results in the $L2HitM$ hardware performance counter being incremented, as the CPU has to invalidate the Level 2 cache line which contains A on C_2 and bring the modified values over from C_1. The cache coherency is maintained by an on-chip CPU cache coherency protocol [40].

Next, we can also calculate the performance impact of address translation misses (TLB Misses), as shown below.

let $x_\Delta = x_t - x_{t-1}$ *for any* x:

$$TLBImpact = \frac{30 * MemLoadRetiredDTLBMiss_\Delta}{CpuClkUnhaltedThread_\Delta}$$

Here, we make use of hardware performance counter $MemLoadRetiredDTLBMiss$, which represents retired memory loads that miss the DTLB (translation look-aside buffer for data). According to Intel, this can take on average 30 additional cycles.

Lastly, we can extend the same principle to the impact of page fault traps, assuming no disk accesses are caused by a page fault. Back in 2003, Haeberlen and Elphinstone measured the single page fault overhead to be around 1700 cycles [62] while in his recent online post [2] Linus Torvalds estimated this to be around 1050 cycles, which is the number we have used in our study as well.

let $x_\Delta = x_t - x_{t-1}$ *for any* x:

$$HPFImpact = \frac{1050 * HardPageFault_\Delta}{CpuClkUnhaltedThread_\Delta}$$

In this last formula, the $HPFImpact$ represents a performance impact of the page faults that have occurred in a particular period of time and calculated by counting $HardPageFault$ which can be done through operating system event collection mechanisms, such as Windows Event Tracing.

[2] https://plus.google.com/+LinusTorvalds/posts/YDKRFDwHwr6

6.6 DATA MODELLING

The next step in our analysis was to create a flexible data model that would allow us to build various visual support tools for developers of parallel programs. Based on our interview analysis, we set out to design a data model that would accommodate the following constraints:

- **Support for the orchestration metaphor.** As one of the major implications for design was the widespread usage of orchestration models, our data storage system should also accommodate the data in a way that facilitates the delivery and implementation of tools that consume the data and visualise it with a consistent metaphor.

- **Support active experimentation.** As we have noticed during our interview analysis, active experimentation (trial and error) is the most common technique used by developers to understand the performance within programs. Based on this conclusion, we set out to design a data model that also supports the process of active experimentation.

- **Support for consistency.** Many of the developers we interviewed mentioned that organisational constraints such as time, make it difficult for them to learn new tools every time they have a particular problem to solve. This requires us to make a data model that can provide data in a consistent way, effectively smoothing the eventual learning curve.

The solution we proposed for storing our data complies with all the constraints and provides a flexible and consistent way to explore, extract, modify and build upon. It is called the SWARM model, which stands for “Scale”, “Worker”, “Action”, “Resource” and “Metric”, and provides support for faceted search over time-series of orchestration model-supported data:

- **Scale.** Represents the scale, or the scope of the event. This is a facet with values within an ordinal class. A scale facet could take following non-exhaustive values: “Machine”, “Processor”, “Program”, “Thread”, “User”. Within a given scope, a particular set of workers operate, for example, if the scale dimension is

set “Program”, each worker is an individual program and the data is automatically aggregated to accommodate for this.

- **Worker.** Represents one or particular worker instances. For example, a particular Core, a program or a particular thread.
- **Action.** This represents the conceptual action that a worker performs on a resource. For example, a worker can be “using” a central processing unit to calculate something, or it can be “allocating” some memory resources for later use.
- **Resource.** The resource that can be used by a particular worker or several workers. To give you an example, a resource facet could take non-exhaustive values such as “CPU”, “GPU”, “Memory”, “Hard Disk”, “Network Interface”.
- **Metric.** Finally, metric represents a type of possible measurement. For example, given that the scale is Program, worker is Program 1, action is “allocating” and resource is “memory”, metric could be any value that is measurable by hardware counters, operating system events or any other direct or derivative means, such as “Allocation speed”, or “Bytes allocated”.

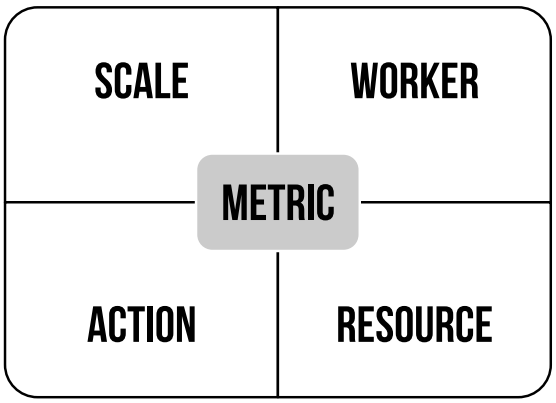

Figure 6.10 – Five facets of the SWARM data model used to store the performance data for the delivery to the client applications and post-processing transformation.

In the SWARM model, we store the data in a way that accommodates for the orchestration model using a time-series of 5-facet data points, as shown in the Figure 6.10. A faceted search system is a common solution for exploratory search problems, which supports active experimentation (or data exploration). Essentially, when a programmer visualises and explores the performance data, it is an exploratory search

problem, which is a specialisation of information exploration which represents the activities carried out by searchers who are either:

- Unfamiliar with the domain of their goal (i.e. need to learn about the topic in order to understand how to achieve their goal).
- Unsure about the ways to achieve their goals (either the technology or the process).
- Unsure about their goals in the first place.

As part of our working programme, we completed a basic performance analytics system, leveraging the performance data model that we have designed based on the interview analysis. This performance analytics system is shown in the Figure 6.11.

Our basic performance analytics system presents a series of horizon graphs, depending on the selected parameters. Horizon graphs increase data density while preserving resolution. While horizon graphs may require learning, they have been found to be more effective than standard line and area plots when chart sizes are small [68].

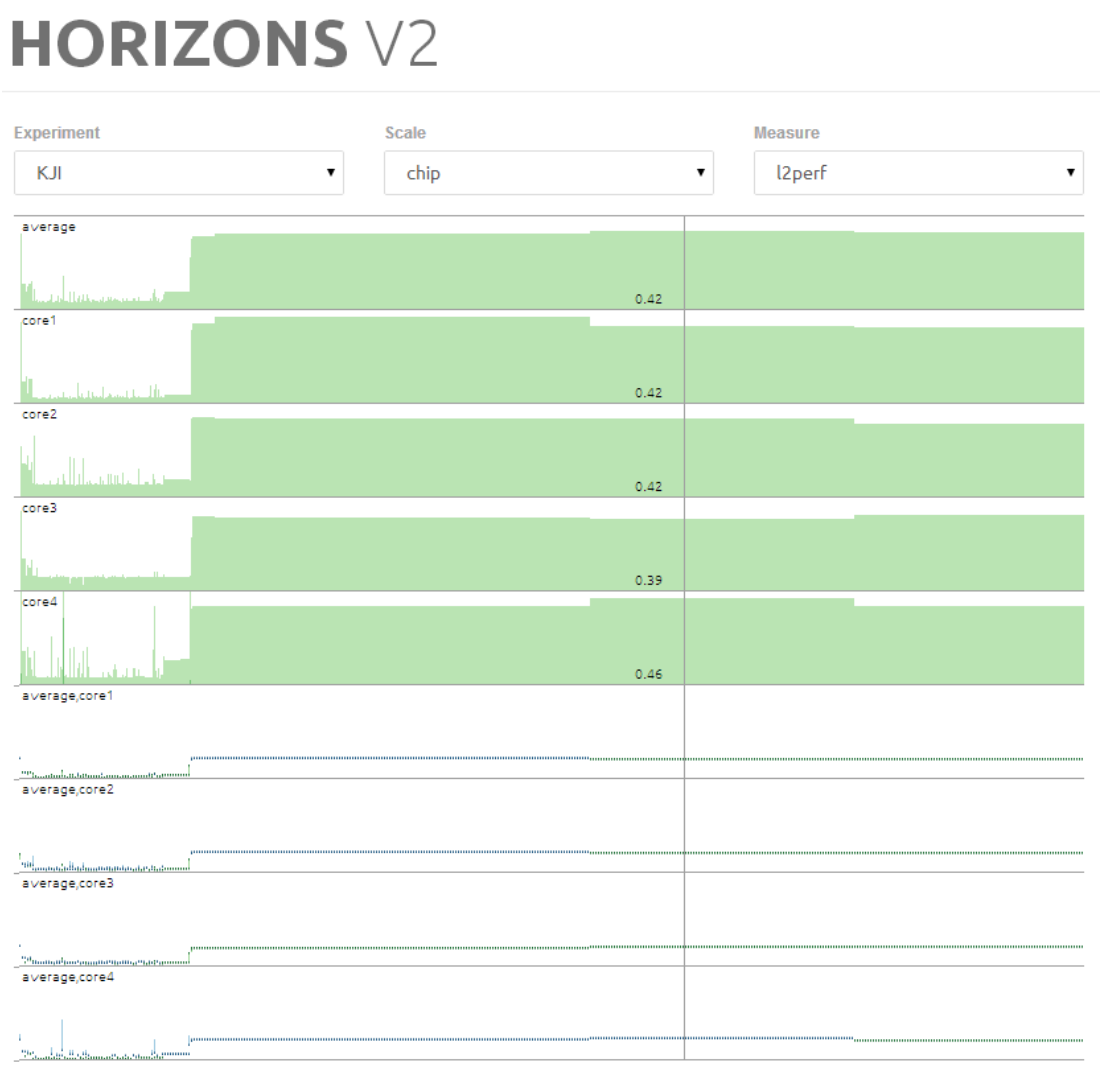

Figure 6.11 – Horizons, a basic performance analytics system that leverages SWARM model as its underlying data provider.

6.7 DATA PROCESSING SYSTEM

In order to lay the foundation for designing various prototypes, we then began to design a data processing system, based on the SWARM model we described above. The system we implemented was designed with the help of traditional extract, transform, and load pattern (ETL). This pattern is widely used in data-warehousing and consists of three steps:

1. **Extract.** The processes related to the extraction of data from external data-sources. We extract data from hardware performance counters and operating system events on the target machine, (i.e. the machine that has a software to monitor the per-

formance of).

2. **Transform.** The processes related to the transformation of the data in order to fit the operational needs. We transform the raw data into SWARM-modelled data structures.

3. **Load.** The data is loaded to a local, non-distributed NoSQL database and indexed for an efficient delivery in the JSON format in order to be consumed by a visual analytics tool via a HTTP REST API.

Figure 6.12 presents our cloud architecture and its ability to horizontally scale, as many more virtual servers can be added. Each individual server in our cloud deployment has the ability to process and deliver the required data for one or several experiments. The system can be scaled automatically based on the amount of experiments running concurrently.

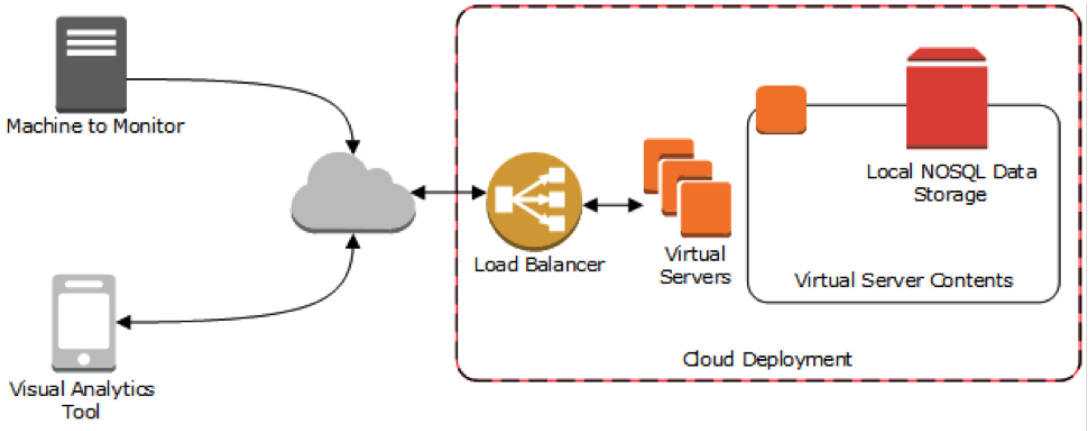

Figure 6.12 – The architectural schema presenting the cloud-based data processing system we built as a common foundation for data storage and delivery.

There is a clean split between different components of the overall architecture. To illustrate the data flow process, consider the following scenario of how the system works from the beginning of data collection to the rendering of the visualisations:

1. On the target machine, a small software is installed. This software integrates the data collection mechanisms in order collect hardware counters and operating system events. When the performance analysis session is initiated by the programmer, it only requires them to start the program, and stop it when they wish to stop the performance analysis session.

2. During the session, the data collection software sends the data to one of our virtual servers, on a specified frequency (by default every 1 second). This is accomplished through HTTP protocol by sending a POST request to the remote endpoint.

3. The incoming data is then stored in the initial format by the server, and processed as it comes in, continuously in the SWARM-compliant format. The data points are resampled, averages and sums are created and the data is populated.

4. Finally, when a visual analytics tool requires a particular set of data, it issues a HTTP GET request to the virtual server for the particular performance session (i.e. experiment) and the server returns the requested data. The data is delivered to the tools via a standard REST API, where the HTTP endpoints are read-only and the parameters can be specified in the url itself. For instance, issuing a request to `http://api.rpc.io/x1/process/program1/cpu/usage/ticks` will yield a timeline for the performance session `x1`, with scale defined as `process`, the worker as being `program1`, resource as being `cpu`, action as being `usage` and the measure defined as `ticks`, which we used to present a sampled CPU usage. The server then returns a JSON-formatted time series of values for the specified parameters, such as:

```
[{"Time":"\/Date(1368403200054+0000)\/","Value":0.123},
{"Time":"\/Date(1368403200055+0000)\/","Value":0.271},
{"Time":"\/Date(1368403200056+0000)\/","Value":0.9615},
{"Time":"\/Date(1368403200057+0000)\/","Value":0.9635}]
```

5. The tool then renders the information where the rendering process is specified within the tool itself, allowing us to build prototypes that mix and match different information sources. Our initial prototype simply renders the time series as a horizon chart, as depicted in Figure 6.13.

Figure 6.13 – A single horizon timeline, representing a rendered time series of CPU usage events for a single program on the process scale.

6.8 CONCLUDING REMARKS

The problem of data locality induced by the 'memory wall' is intimately intertwined with parallelism [84]. When developers build parallel software, performance is usually one of the key goals, yet data locality is often just as important as parallelism for performance. Thus, the programmer needs to be able to identify data locality problems when they arise in parallel programs. In addition, parallel threads executing on different cores often share the same data in one or more levels of cache which can improve locality. But equally, the threads may end up competing to keep their own data within the cache. The result can be complex interactions that cause non-obvious locality problems that may be difficult for the developer to identify.

In this Chapter we have examined the data locality problem, both its architectural and its algorithmic causes, along with some of the measurable counters and events that can be collected in order to help with the diagnosis of poor data locality in parallel software.

We have analysed two matrix multiplication algorithm implementations which can be used to illustrate both good and poor data locality.

We have shown that simple counters such as cache misses and cache hits can be used together with some metrics to effectively approximate the impact of individual events on the performance of the program. A good starting point proved to be the metrics provided and used by Intel in their tools; we have extended the concept of expressing the performance impact in terms of cycles lost to other components, such as TLB and page fault traps raised by the operating system.

Going forward, we have implemented a data collection utility which allows attributing various hardware performance counters to individual threads, enabling us to "assign blame" to various threads in the program for counters such as cache or TLB misses.

Lastly, we have constructed an ETL delivery mechanism with a standardised schema (SWARM Model) for storing performance measurements in a time-series fashion with enabled faceted search.

The next chapter covers the design and evaluation of a tool we have designed based on our insights, models and analysis presented here and in the previous chap-

ters.

CHAPTER 7 VISUALISING THE PERFORMANCE

There are aspects of data locality that are poorly understood by many developers simply because they are related to low-level aspects of modern computer architectures. For example, modern processors have a special type of cache, known as the translation lookaside buffer (TLB), that is used to store parts of the page table from the virtual memory system. Many of the same issues arise with TLBs as with caches in parallel software. However, it is particularly difficult for a developer to identify a performance problem if they are unaware that it can arise. Nonetheless, it should be possible for developers to identify and solve problems without becoming experts in the low-level details of the multi-core architecture.

7.1 VISUALISATIONS TO SUPPORT DATA LOCALITY ANALYSIS

We address the problem of diagnosing data locality by creating an information visualisation (InfoVis) system. InfoVis can be described as a cognitive activity in which users engage, with the potential of gaining an understanding and an insight from data represented by a visual medium [154]. As well as this perceptual and cognitive element, users are also involved in an interaction with the visualisation tool, and hence the entire experience can be seen as being based both on conceptual models of the domain and on instantiated mental models of the visualisation interface. Both activities are cognitively interconnected and difficult to separate, making it challenging to evaluate, as the whole experience needs to be evaluated. This represents a major challenge from a Human-Computer Interaction perspective [48].

It has previously been suggested that visualisations can effectively support developers in the task of software maintenance by relating information more efficiently, presenting relevant cues to the programmers so that they can accomplish the mainte-

nance task at hand [87].

The process of program optimisation can be viewed as an exploratory search problem; this view is helpful where the users (programmers) lack the knowledge or contextual awareness to formulate precise queries or navigate complex information spaces (i.e.: performance of their programs), the search task requires browsing and exploration, or system indexing of available information is inadequate [170]. All of these are potentially applicable to the task of optimising parallel programs, and hence we are interested in multi-faceted visualisations that support data exploration in a interactive manner.

In this chapter, we present a data locality performance analyser, namely **Data PAL** : an interactive visualisation tool designed with the objective of allowing programmers to easily identify the presence of data locality issues. It is intended to support identification of problems along several dimensions:

- **Process/Run.** As the first question in the decision tree underlying the visualisation, a programmer might ask whether there is a potential data locality issue with the current process or experimental run. Figure 7.1 depicts the part of the visualisation responsible for this.

- **Time.** A timeline visualisation (Fig. 7.2) part of the tool allows the programmer to identify where in time the problem might occur. Different phases of a program are likely to have different properties with respect to locality.

- **Thread.** A thread view (Fig. 7.3) allows programmers to identify threads exhibiting poor data locality.

The design and empirical evaluation of the tool allows us to explore the design space and potential impact of this form of programmer support, and provides some interesting insights into the specific example of data locality within software engineering practice.

The graphs are based on data from hardware performance counters combined with constants and formulas (e.g.: clock cycles wasted due to L2 cache misses, etc.) publicly provided by the microprocessor manufacturer (Intel in this case). These calculations are used to transform the huge number of available hardware performance counters to a set of simple cost-functions. These performance metrics are intended to aid devel-

opers in the process of forming mental models of the performance of the program; we define mental models as knowledge instantiated in working memory for the particular need and context. In turn, this helps developers to reason about the program and form hypotheses on possible performance bottlenecks and on the possible solutions to them.

7.1.1 GREENLIGHT VIEW

An analysis of the process of diagnosis of data locality problems yields an initial problem of deciding whether a program has a potential issue with data locality. A top level visualisation, such as that depicted in Figure 7.1 can aid in this initial step. We term this the "Greenlight View". The visualisation consists of a stacked bar chart, representing the estimation of the proportion of time (CPU cycles) spent executing or loading data from various underlying components of the architecture. The rationale for the design builds on a previous interview study conducted with programmers:

- **Purpose 1**: Provide enough detail for developers to discriminate between performance components.
- **Purpose 2**: Allow developers to easily compare the proportion of performance components from different program runs (or program versions).

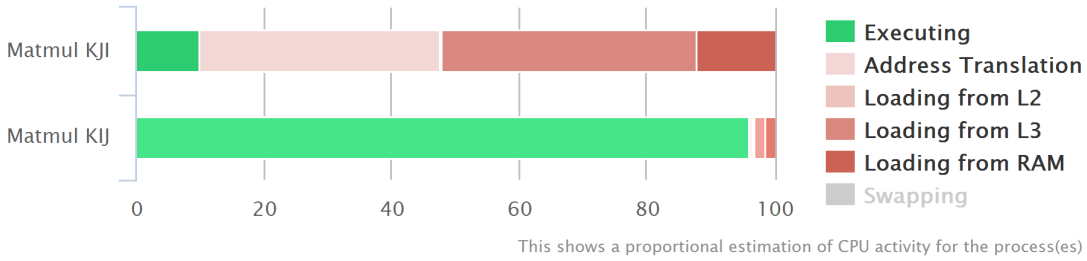

Figure 7.1 – Greenlight View for global performance assessment.

7.1.2 TIMELINE VIEW

Real programs are likely to have different phases of execution. Even within the same code, data locality can change over time due to fragmentation or as different parts of a

large data set are processed. Thus a timeline view can be helpful for identifying time intervals where a data locality issue might be present.

Depicted in Figure 7.2 is the "Timeline View" of the **Data PAL** visualisation. This visualises the changes to the contributions of the components within the Greenlight View over time. This is done using a stacked area chart and having the y-axis represent wall clock time (elapsed time) of the execution of the program.

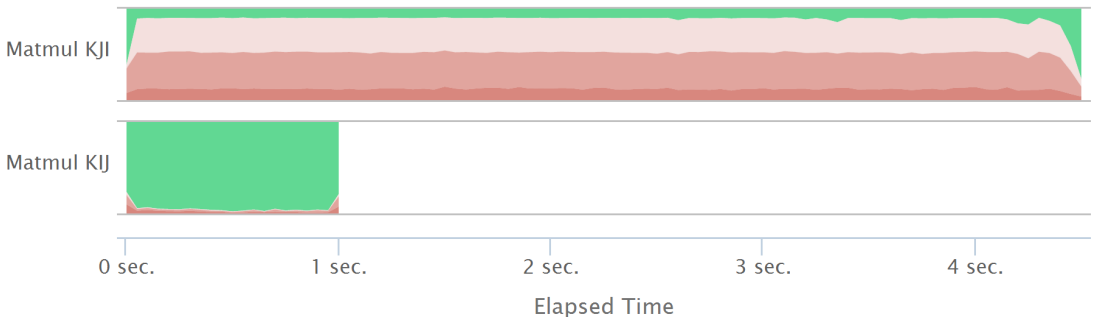

Figure 7.2 – Timeline View supports identification of time intervals where a data locality issue might be present

- **Purpose 1**: Compare different finite algorithm implementations by elapsed time (shorter is better).
- **Purpose 2**: Support the visualisation of program phases and determine a performance diagnosis window.
- **Purpose 3**: Provide a format in which events can be annotated (e.g. entering a particular loop or method).

7.1.3 THREAD VIEW

Depicted in Figure 7.3 is the "Thread View" visualisation which represents the different threads using parallel coordinates (a technique for visualising multivariate data [78]).

- **Purpose 1**: Provide a way to identify poorly performing individual threads or groups of threads.
- **Purpose 2**: Identify expected or unexpected outliers and clusters.

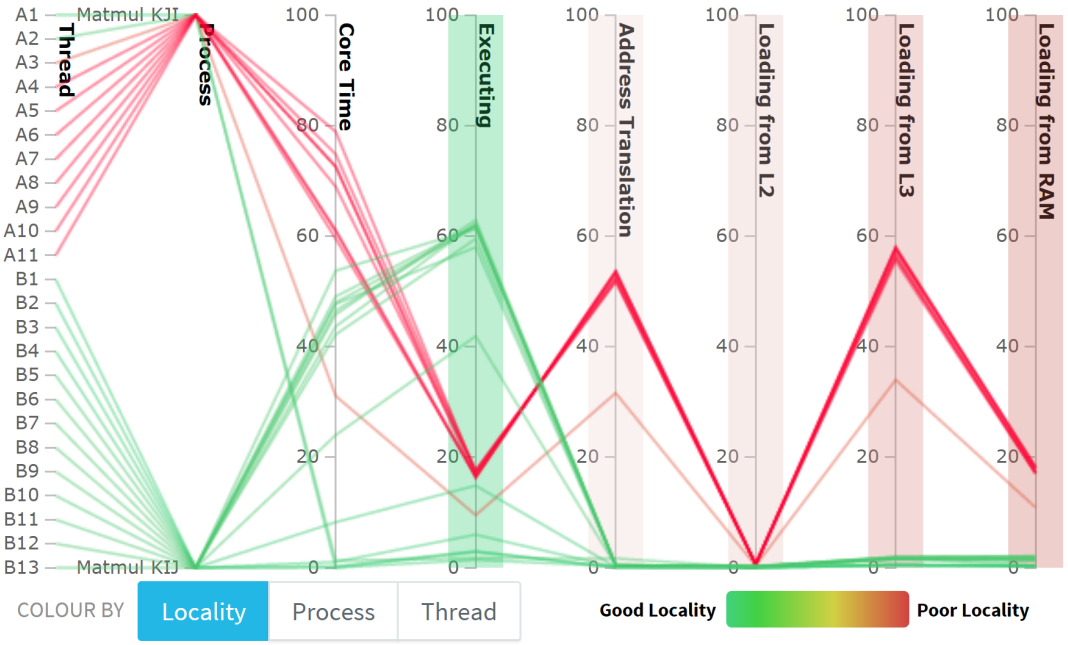

Figure 7.3 – Thread View for identifying threads exhibiting poor data locality symptoms.

- **Purpose 3**: Identify threads which occupy a greater proportion of core time, which also have poor data locality (Purpose 1), and hence provide more potential benefit for optimisation.

The visualisations are interactive, and allow different data elements (TLB, L2, L3) to be added or removed from the visualisation, subsets of the threads (or individual threads) to be selected, and so on.

7.2 EXPERIMENTAL DESIGN

As a quick reminder, the last research question that we have set out to answer in this thesis is **RQ 3: To what extent can a visualisation effectively support programmers in the task of optimising parallel programs?** In the section below we present and discuss several sub-research questions underlying our experiment and its design.

7.2.1 RESEARCH QUESTIONS AND POTENTIAL FORMATS

RQ 3.1: Does the visualisation allow developers to identify the presence of data-locality issues?

While this question at the first glance seems a like simple "yes or no" question, the answer depends on various intrinsic and extrinsic variables some of which are very difficult (if even technically possible) to measure. Ultimately, the developer is the one who will be able to answer the question within a given context, as we discussed previously. There are several possibilities that need to be considered in order to successfully answer this question:

- **Threshold-based evaluation.** The simplest way to approach this issue would be to set a threshold on some measurable variable, for example a threshold of 10%. In that case, we would judge a data locality problem to be present if at least 10% of total wall clock time of the program was spent waiting for data to arrive from L1, L2, L3 or Main Memory. While this seems reasonable, it poses several problems: (a) the results of this evaluation would be of questionable generalisability, since the threshold itself depends on the program, context, various goals the programmer is trying to achieve and many more constraints, for example, 10% may be for too high for a console game developer processing millions of game entities a frame, while for another context this could be acceptable and not worth the programming effort involved in optimising further; (b) the actual threshold is difficult to choose; there may be different methods for determining the appropriate threshold, such as an oracle-based approach (experts or a numerical measure), a pilot study, or a pre-study focusing entirely on figuring out what the threshold value should be, each of which will still be context dependent.

- **Comparison-based evaluation.** Another approach would be indirectly observing the change of hypothesis between different representations of the program. For example, we can show to the developer the same program with and without the visualisation, without assuming any threshold. Each time, the programmer would simply be asked to identify the presence of data locality issues and potentially to assess the severity (eg: "critical" , "important", "moderate", "low" or "insignificant"). The benefit of this approach is that we are not required to set a particular threshold, but on the other hand it requires careful setup of the experiment and will yield a mixture of quantitative and qualitative results which are more open to interpretation.

An experimental format based primarily on the comparison-based evaluation ap-

proach was used, although an oracle (a domain expert working with the aid of the visualisation) was used for the correctness criterion. We created several pairs of programs designed to accomplish the same or a very similar task with implementations that differ in performance characteristics when it comes to data locality. A source-code only control was chosen. A comparison based format could also be used to investigate the relative performance of different forms of tool support. As we are aiming to investigate the impact of providing this form of support, the clearest comparison is with the unsupported case, rather than with simpler or more complex supports (for example, showing large numbers of different raw hardware performance counters), or alternative representations of data (such as tabular representations).

RQ 3.2: Does the visualisation help to reduce cognitive load of the data locality identification task?

As mentioned above, 66% of the developers surveyed do not use any concurrency tools, as illustrated by a quote from our field study: *I think the only way I can do most of this debugging is stare at it and debug it in my head, which is the least effective way you can ever debug.*

This "debugging-in-my-head" technique requires a lot of mental effort (cognitive load) and sometimes takes hours, as one of our participants mentioned. To gain a better understanding of how a visualisation of data locality helps to reduce cognitive load and make the task at help seem easier to programmers, we asked participants about the perceived difficulty of the task, ranging from *very easy* to *very difficult*. Additionally, we also added a confidence measure on each question we asked our participants during the experiment, ranging from *very confident* to *very doubtful*, with the hypothesis that the perceived difficulty and confidence might correlate and if a task is perceived as "very easy" using a visualisation, the programmer might also be very confident of their answers.

RQ 3.3: Do the potential benefits of a data locality visualisation depend on the level of experience of the programmer? Experts usually have significantly better initial hypotheses, better mental models of what might be happening within the program, and are also substantially better and faster in finding bugs [60]. With this experiment, we wanted to see whether **Data PAL** allows novice programmers to improve their performance with respect to expert programmers when it comes to understand-

ing performance overhead related to poor data locality, or whether experts are still vastly better than novices even with the visualisation tool at their disposal.

In order to answer this question we employed a mixture of qualitative and quantitative analysis. For the qualitative analysis we have asked the participants with and without the visualisation to explain the problem, which would allow us to get an insight in the change of hypothesis of both novices and experts. For the quantitative analysis, we were interested how the confidence in the answers (as an interface for participants' confidence in their internal hypotheses) of novices and experts changes when they use a visualisation.

7.2.2 METHODOLOGY

The study was designed following the rationale described in section 7.2.1 above and conducted via an online web-interface. The questions and problems posed were refined iteratively through a pilot study involving 3 participants in which users were observed in person using a think-aloud protocol. The results from the pilot study were not included in the final experimental data.

Through the pilot study we were able to estimate the total duration of the experiment which was around 40-50 minutes; this is relatively demanding on participants and impacts on recruitment and completion of the full experiment. One of the participants of the pilot study took 1 hour 20 minutes to complete the experiment. In the full study, 33 participants completed the entire experiment and we present an analysis of the results below. We did not include any partial results in our analysis for the sake of fair comparison.

The experimental design took guidance from the work done by R. Wettel and M. Lanza [168, 169], who conducted an extensive survey of research dealing with experimental validation of software engineering, InfoVis and software visualisation approaches [140]. A number of principles taken from this served an important role in the design of the experiment and is composed of the following:

1. **Involve industry participation.** A representative sample of software practitioners is important for any study. Efforts were made to recruit industry participants to the study in order to obtain more generalisable results.

2. **Provide tutorial of the tool and problem background.** Data locality is not a trivial problem to deal with and the tool requires a basic understanding of the vocabulary and the problem at hand. Thus, some tutorial material and an example are delivered as part of the preamble of the experiment.

3. **Take into account the level of experience of the participants.** It is well established that experts have generally better mental models and are able to reason about programs faster. As it is possible that expert or experienced programmers might reason more effectively about the performance of data locality, data about experience and self-reported expertise are gathered as part of the experimental protocol.

4. **Provide the same data to all participants.** The observed effect of the experiment is more likely attributable to the independent variables if this guideline is followed [140]. The automated delivery format of the experiment standardised administration of the experiment and information provided to participants.

5. **Use more than one subject system.** Since performing the same experiment with different systems can lead to significantly different results, as demonstrated by Quante [135], we have created a small set of 4 pairs of very different programs for this experiment.

6. **Choose programs representative of the real-world.** Experienced programmers often do micro-benchmarking and isolate pieces of code that require optimisation or parallelisation. For this experiment we needed to chose a set of programs representative of the real-world, while small in size that can be grasped by programmers during a controlled experiment. While this is an unavoidable tradeoff for this type of experimental work, it is also a potential limitation, as we discuss in the Limitations section.

The study was run over a period of several months and used a snowball sampling approach leveraging advertisements to special interest groups and professional social networks, as finding both novices and experts willing to undergo a somewhat cognitively demanding ()approximately 40 min) experiment was a challenge. Recruitment was well balanced between academia and industry participants as participants were recruited from a range of software companies (very large, medium and small) as well

as multiple academic institutions. Four sessions were observed in person (excluding pilot sessions) in order to provide richer data and to allow for follow up on issues of interest. The remainder were completed fully remotely.

Knowledge of compiler optimisations on particular platforms represented a potential confound, and so programs were compiled without optimisation when generating data for visualisation.

The participants were rewarded with a €10 book token for their time. Ethical approval was obtained from the relevant ethics committee. The experiment was conducted in several steps. First, the participants were asked to fill-in background information and were shown a short tutorial/refresher on the impact of data locality to the performance of programs. Then, they were asked to assess two pairs of programs with source code alone. Next, participants went through a short training page depicting and explaining the three components of the visualisation with some animations demonstrating interactive elements such as selecting a single thread, filtering out data, etc. Finally, the participants would go through the remaining two pairs of programs with the visualisation at their disposal. For both source-code and visualisation the participants were asked to answer the following questions:

1. *Which program will finish the execution first?*
2. *Which described program has better data locality?*
3. *How would you describe the data locality of Program A?*
4. *How would you describe the data locality of Program B?*
5. *How challenging was the problem?*
6. *How would you explain the difference in performance between the two programs? (open question)*

Moreover, for the problems answered with the visualisation the participants were asked two following additional questions:

1. *Looking at the visualisation alone, which threads have the best scope for improving performance (i.e. where would you start looking)?*

2. *Is there anything about the visualisation which surprises you in any way and if yes, how?*

Participant	Experience	Education	Expertise
P1, Male	Junior	High School	Average
P2, Male	Junior	Bachelors	Above Average
P3, Male	Junior	Bachelors	Above Average
P4, Male	Junior	Bachelors	Average
P5, Male	Junior	Bachelors	Novice
P6, Male	Junior	Bachelors	Novice
P7, Male	Junior	Bachelors	Above Average
P8, Male	Junior	Masters	Novice
P9, Male	Junior	Masters	Novice
P10, Male	Junior	Masters	Above Average
P11, Male	Junior	Masters	Novice
P12, Male	Junior	Masters	Novice
P13, Female	Junior	Doctorate	Average
P14, Male	Junior	Doctorate	Above Average
P15, Male	Junior	Doctorate	Above Average
P16, Male	Senior	Bachelors	Expert
P17, Male	Senior	Masters	Above Average
P18, Male	Senior	Masters	Expert
P19, Male	Senior	Doctorate	Average
P20, Male	Senior	Doctorate	Average
P21, Male	Senior	Doctorate	Average
P22, Male	Veteran	High School	Expert
P23, Male	Veteran	High School	Average
P24, Male	Veteran	Bachelors	Above Average
P25, Male	Veteran	Bachelors	Expert
P26, Male	Veteran	Bachelors	Above Average
P27, Male	Veteran	Bachelors	Above Average
P28, Male	Veteran	Masters	Expert
P29, Male	Veteran	Masters	Expert
P30, Male	Veteran	Masters	Average
P31, Male	Veteran	Doctorate	Expert
P32, Male	Veteran	Doctorate	Novice
P33, Male	Veteran	Doctorate	Expert

Table 7.1 – Participants, with years of experience in the domain (Junior/Senior or Veteran), self-assessed expertise levels in parallel programming and highest education level.

The participants demographic data is shown in the Table 7.1, including several non-standard dimensions such as years of experience in the industry/academia and a self-assessed expertise level in parallel programming. The years of experience was grouped into three distinct categories being junior (0 to 5 years), senior (5 to 10 years) and veteran (10+ years).

7.2.3 TASKS

A set of pairs of programs were designed that try to accomplish the same conceptual goal, but vary in implementation and data access pattern. We used a latin square design for the experiment and presented pairs of programs to participants in different order, and with different types of treatment (source code vs. source code and visualisation).

MATRIX MULTIPLICATION

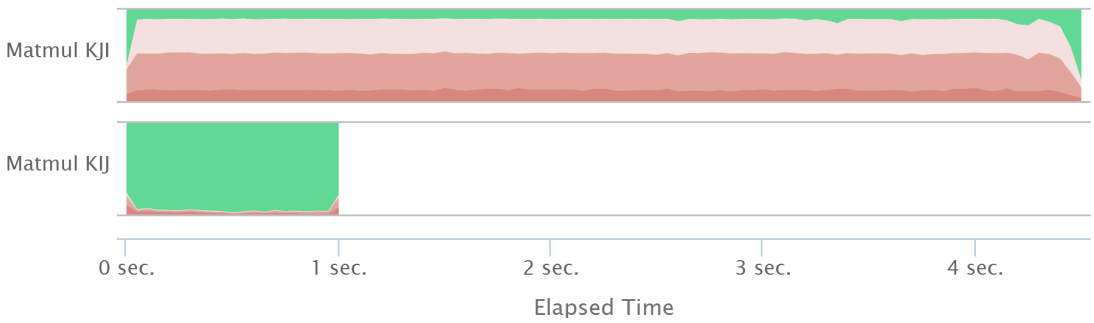

Figure 7.4 – Data PAL timeline representation of performance of two matrix multiplication implementations used for the experiment.

Dense matrix multiplication is a commonly used component of a variety of applications, and a core component in many scientific computations. Matrix multiplication can be implemented in numerous ways, and the task of multiplying large matrices is extremely data-intensive, providing us with a great example of both good and bad data locality patterns. Moreover, there has been a great deal of interest in developing parallel formulations of this algorithm and testing its performance on various parallel architectures [61].

In our evaluation we selected two matrix multiplication algorithms with swapped rows and columns. As the reader can see on the Figures 7.1, 7.2 and 7.3, the second program (ie: Matmul KIJ) is roughly **4.5 times faster** than the first program, which can be explained largely by the impact of L2 cache misses and poorly performing TLB of the first program.

```
parallel_for (0, n, [&](int k) {
   for (int j=0; j<n; j++) {
      int r = b[k][j];
      for (int i=0; i<n; i++)
         c[i][j] += a[i][k] * r;
   }
});
```

Figure 7.5 – Simple program to perform large matrix multiplication used to generate visualisation data - this version has poor data locality.

The source code for the first program, performance of which is shown in the first chart of the Figure 7.4 can be seen in the Figure 7.5. In this program, the 5000 by 5000 matrices `a` and `b` are multiplied in parallel to produce the matrix `c`. The matrices are represented in memory as sequential arrays, however this implementation does not access them sequentially which results in poor data locality and a significant amount of cache misses.

```
parallel_for (0, n, [&](int k) {
   for (int i=0; i<n; i++) {
      int r = a[i][k];
      for (int j=0; j<n; j++)
         c[i][j] += r * b[k][j];
   }
});
```

Figure 7.6 – Simple program to perform large matrix multiplication used to generate visualisation data - this version has good data locality.

The source code for the second program, performance of which is shown in the second chart of the Figure 7.4 can be seen in the Figure 7.6. In this program, the 5000 by 5000 matrices `a` and `b` are multiplied in parallel to produce the matrix `c`. The matrices are represented in memory as sequential arrays and this implementation does access them sequentially which results in good data locality and few cache misses.

PARTICLE SYSTEMS

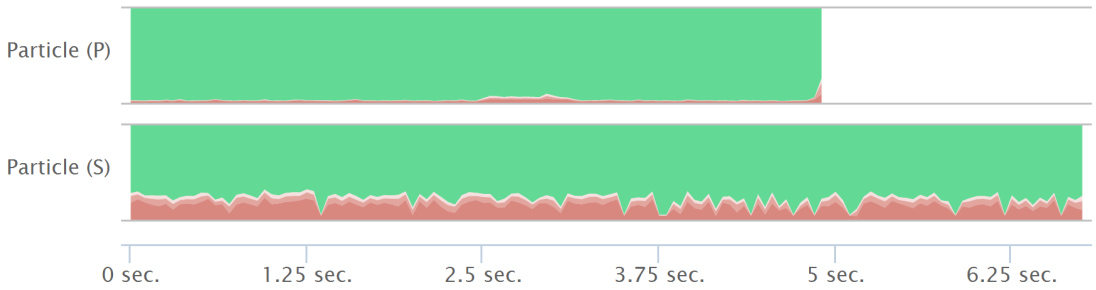

Figure 7.7 – Data PAL timeline representation of performance of parallel and sequential particle system implementations

A common task faced by programmers consists of parallelising an existing algorithm. For this task, we have aimed at a simple and understandable example, and implemented two particle system algorithms: a serial and parallel version. The implementation consists of a data structure representing a single particle that consists of coordinates (x, y, z) and velocities (vx, vy, vz), an update method is invoked on each particle in which the delta-time is provided and the new coordinates are computed.

Figure 7.7 shows not only that the parallelised particle system is slightly (1.4x) faster but also has better data locality performance, proportionally. The speedup itself is rather small, as the program also exhibited, by design, a task-granularity problem. This was done with the intent of keeping a good balance between "toyness" of the examples and the reality, where performance problems are often interleaved and interdependent.

```
struct Particle {
   float x, y, z, w;
   float vx, vy, vz, vw;
};

Particle *particles;
int count;

void update(float dt) {
   parallel_for(0, count, [&](int i) {
      auto p = particles[i];
      p.x += p.vx * dt;
      p.y += p.vy * dt;
      p.z += p.vz * dt;
      p.w += p.vw * dt;
   });
}
```

Figure 7.8 – Simple program to process several million particles used to generate visualisation data - this version is parallelised.

The source code for the first program, performance of which is shown in the first chart of the Figure 7.7 can be seen in the Figure 7.8. In this program, the array of several million `Particle` data structures is updated in parallel, using the `parallel_for` construct from C++ parallel patterns library provided by Microsoft for Visual C++.

```
struct Particle {
    float x, y, z, w;
    float vx, vy, vz, vw;
};

Particle *particles;
int count;

void update(float dt) {
    for (int i = 0; i<count; i++) {
        auto p = particles[i];
        p.x += p.vx * dt;
        p.y += p.vy * dt;
        p.z += p.vz * dt;
        p.w += p.vw * dt;
    }
}
```

Figure 7.9 – Simple program to process several million particles used to generate visualisation data - this is a serial version.

The source code for the second program, performance of which is shown in the second chart of the Figure 7.7 can be seen in the Figure 7.9. In this program, the array of several million `Particle` data structures is updated in sequentially on a single thread.

ACCOUNT UPDATE

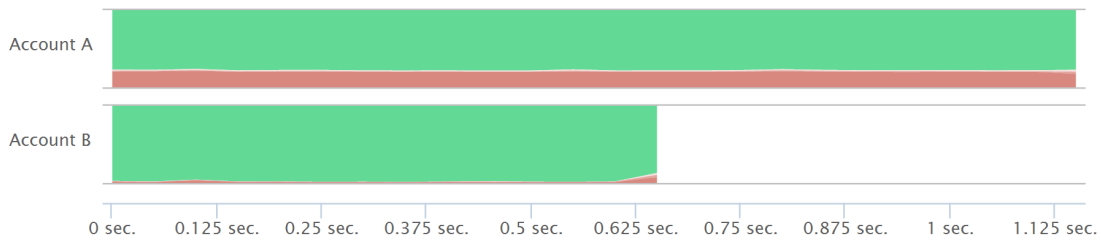

Figure 7.10 – **Data PAL** timeline representation of performance of two in-memory database schemas

The third task draws a parallel between database and object-oriented design where we have two relatively straightforward schemas of user accounts that include fields like name, address, user id and reputation. The experiment consists of a large number of accounts being updated in parallel, with an update that consists of incrementing the reputation. This is described to the participants as a "daily update". Timeline representations are depicted in the Figure 7.10, where **Account B** is almost 2x faster than **Account A**, due to better spatial locality of reference as the schema splits unused values, namely address and name of the user into another structure that is referenced by pointer to avoid needlessly filling the cache. In fact, the second implementation has **18.2x** less L3 cache misses than the first one.

```
struct Account {
  long id;
  char name[50], address[300];
  float reputation;
}

void update(){
  parallel_for(0, count, [&](int i) {
    accounts[i].reputation += 2.5;
  });
}
```

Figure 7.11 – Simple program to process several million user accounts used to generate visualisation data - this version has poor data locality.

The source code for the first program, performance of which is shown in the first chart of the Figure 7.10 can be seen in the Figure 7.11. In this program, the `Account` data structure contains the fields `name` and `address` which are directly embedded and when the data structure is loaded in memory to update the `reputation` field, the contents of all the fields are loaded into processor cache.

```
struct AccountInfo {
  char name[50], address[300];
};

struct Account {
  long id;
  AccountInfo* info;
  float reputation;
};

void update(){
  parallel_for(0, count, [&](int i) {
    accounts[i].reputation += 2.5;
  });
}
```

Figure 7.12 – Simple program to process several million user accounts used to generate visualisation data - this version has better data locality

The source code for the second program, performance of which is shown in the second chart of the Figure 7.10 can be seen in the Figure 7.12. In this program, the the `Account` data structure is split between `Account` and `AccountInfo`. The latter contains the fields `name` and `address` and the first one embeds a pointer to the latter. When the `Account` is loaded from memory to the cache, it only loads a single pointer to the name and address which results to a better data locality.

PHASED LOOPS

The fourth task aims to visualise different program phases by doing some floating point and integer multiplications respectively, split into two "parallel for" loops where the first loop has a large stride which results in poor data locality pattern which can be seen in the Figure 7.13. The program where the data type on which some multiplication is performed is a 32-bit floating-point number takes slightly more time and has significantly more L1, L2 and L3 cache misses, around 3 times more than its 32-bit integer counterpart.

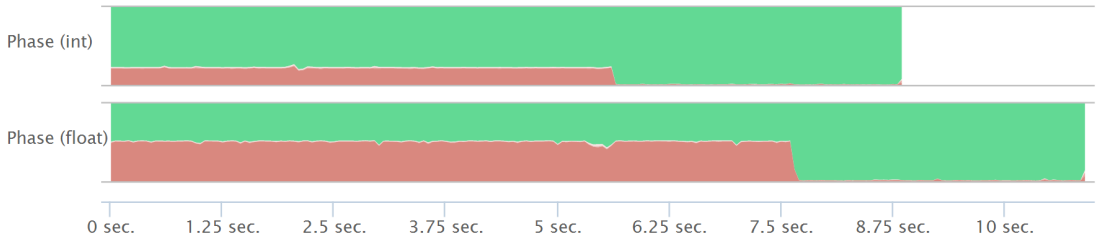

Figure 7.13 – Data PAL timeline representation of performance of a pair of programs consisting of two loops each

```
void update(float* vector){
    for (int m = 0; m < 1000; m++){
        parallel_for(0, count, 100, [&](int i) {
            vector[i] = vector[i] * m;
        });
    }

    for (int m = 0; m < 10; m++){
        parallel_for(0, count, [&](int i) {
            vector[i] = vector[i] * m;
        });
    }
}
```

Figure 7.14 – Simple program to illustrate different program phases - this version uses floating-point numbers.

The source code for the first program, performance of which is shown in the first chart of the Figure 7.13 can be seen in the Figure 7.15. In this program, two parallel loops are computed where first loop goes through every 100th element of the vector and the second loop goes through every element in the vector. In total, both loop perform the same amount of operations: 10 times the vector size.

```
void update(int* vector){
    for (int m = 0; m < 1000; m++){
        parallel_for(0, count, 100, [&](int i) {
            vector[i] = vector[i] * m;
        });
    }

    for (int m = 0; m < 10; m++){
        parallel_for(0, count, [&](int i) {
            vector[i] = vector[i] * m;
        });
    }
}
```

Figure 7.15 – Simple program to illustrate different program phases - this version uses integers.

The source code for the second program, performance of which is shown in the second chart of the Figure 7.13 can be seen in the Figure 7.14. In this program, a floating-point computation is performed which results to more costly CPU instructions while keeping the size of the data same: 32-bit per value.

7.3 RESULTS

In order to analyse the results of the experiment, we adopted a hybrid qualitative and quantitative approach. Wherever possible we investigated the results using statistical analysis, and enriched our understanding of the domain using abundant feedback, comments and answers to open questions gathered during the experiment. This section describes some of the important results observed during the experiment.

CORRECTNESS

An independent factorial ANOVA was conducted to investigate the effect of the visualisation on the evaluation of *correctness*. Correctness was determined by an oracle, a domain expert in parallel programming who examined the code with the help of the visualisation.

As shown in the Figure 7.16, there was a significant difference in *correctness* between the answers using only the source code (M = 52.50, 95% CI [44.76,60.24], SD = 49.94, n = 160) and the answers using source code together with the visualisation (M = 83.97, 95% CI [77.69,90.25], SD = 36.69, n = 131), F(1, 295) = 10.61, p = .0013. However, the interaction effect between the type of treatment (source code vs. visualisation) and *experience* of the participants was not significant, F(3, 295) = 0.27, p = .8483.

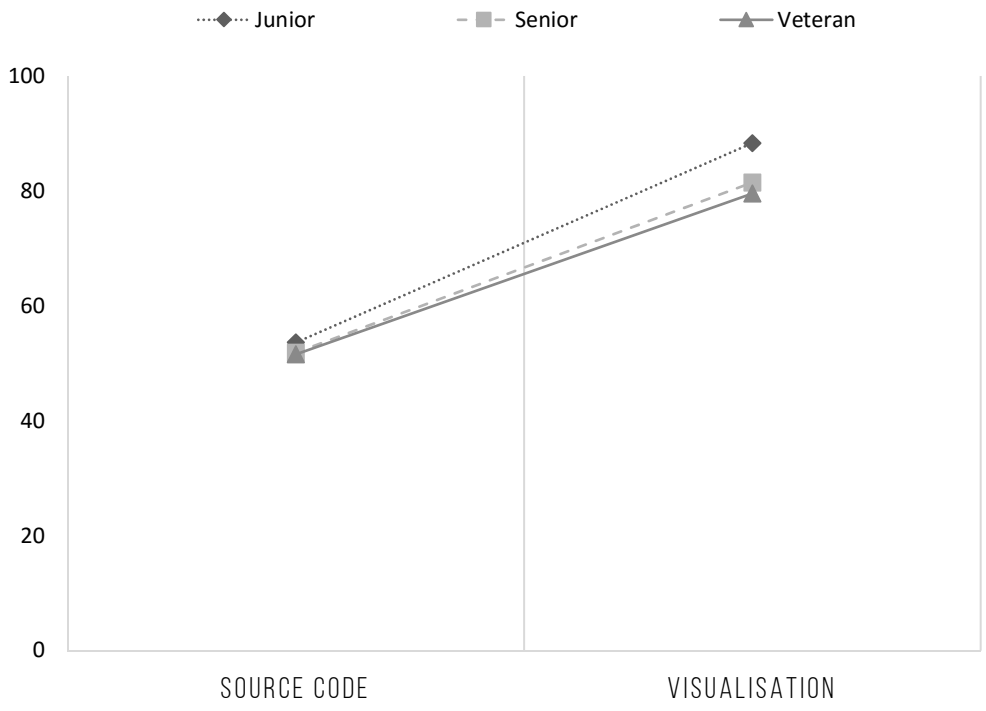

Figure 7.16 – Independent Factorial ANOVA for 2-way interaction of treatment type and experience on correctness

CONFIDENCE

We looked at the effect of participants experience, categorised as junior, senior or veteran programmers based on the years of professional experience. To compare the effect of both visualisation and experience as well as their interaction on the participants' *confidence in their answers*, an Independent Factorial ANOVA was conducted. There was a significant difference in confidence between the participants who used only the source code to answer the questions (M = 0.86, 95% CI [0.75,0.97], SD = 1.02, n = 335) and the participants using the visualisation tool (M = 1.21, 95% CI [1.12,1.30],

SD = 0.81, n = 327), F(1, 396) = 10.91, p = .001.

In other words, when using the visualisation, participants rated their answers with higher confidence than just by looking at the source code. Comparing junior programmers (M = 0.75, 95% CI [0.59,0.91], SD = 1.08, n = 172), senior programmers (M = 1.26, 95% CI [1.09,1.43], SD = 0.74, n = 69) and veteran programmers (M = 1.28, 95% CI [1.15,1.41], SD = 0.82, n = 161), a **significant main effect on confidence was determined**, F(2, 656) = 21.42, p = .001. These results can be seen in the in the Figure 7.17 where the confidence ratings of both junior and senior programmers are significantly higher when using the visualisation, while veteran programmers show no significant difference between the two conditions.

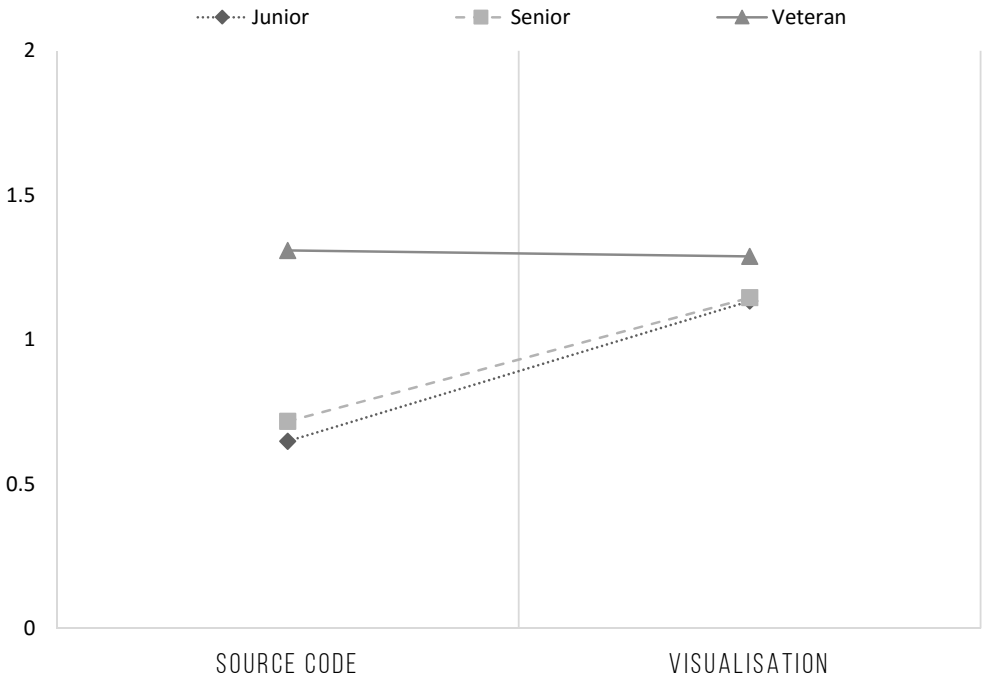

Figure 7.17 – Independent Factorial ANOVA for 2-way interaction of treatment type and participants' experience on the self-assessed confidence level of the answers.

Perceived Difficulty

A Kruskal Wallis test was conducted to investigate the effect of the treatment (pair of programs) on perceived difficulty as reported by participants. This difference was **not significant**, $X^2 = 1.95$, p = 0.582. Additionally, we performed tests to see whether

experience or self assessed expertise as any effect on the perceived difficulty, however, no statistically significant differences were found.

7.4 QUALITATIVE ANALYSIS AND DISCUSSION

In this section we discuss the results in the context of our research questions and incorporate qualitative data from participants responses, post-experiment interviews and think aloud pilots.

RQ 3.1: Does the visualisation allow developers to identify the presence of data-locality issues? Unsurprisingly, our results show that the participants' correctness has dramatically increased when they used the visualisation. The correctness was evaluated on two questions regarding the identification of "first program to finish execution" and the "program with a better data locality pattern".

During the experiment, participants were asked to comment on surprises observed while using the visualisation, below we detail a number of recurrent themes that surprised our participants, both experts and novices.

Some programmers under-estimated the importance of data locality on the performance of the programs. Many of the participants were surprised by how much time a particular program actually spent loading from memory.

> **Veteran**: "*Seems to be a lot of time spent loading from RAM. I expected A to be worse, but maybe not that much worse.*"

> **Veteran**: "*I'm surprised B spends so much more time loading from RAM. It would be interesting to see at what iteration count the overall elapsed time for B is less than A. i.e. at what point the overhead of a multi-threaded model is not worth it.*"

Some programmers over-estimated the effect of parallelisation on the performance of the program, especially in the particle system programs, depicted in Figure 7.7 where the parallel version also contains a task granularity problem.

> **Veteran**: "*(...) the fact that the parallel program is not even 2x the sequential program.*"

Some of the surprises among experts are quite interesting, as it suggests that experts have more complex mental models when it comes to parallel architectures such

as hyper-threading and compiler optimisations of parallel loops.

Junior: "*I would have expected hyper-threading to help more in program B by hiding memory access latency, but it seems that the memory is able to keep up with the core, resulting in some threads getting little core time.*"

In particular, the participant below was actually able to diagnose one of the performance problems where the compiler did not optimise a loop.

Senior: "*It is very surprising that Program B spends so much more time loading data from memory than Program A. This indicates that the compiler was not able to optimise a very simple 1D loop where the data is laid out in array of structures format.*"

Programmers were also able to identify specific threads using the parallel coordinates visualisation, which is useful for heterogeneous programs where threads might contain different workloads (e.g. a game engine) and the ability to identify such threads is important:

Junior: "*Last threads (14, 13, ...) of parallel program are more active than first ones (1, 2, ...). It would be nice if first threads could be more active.*"

Senior: "*The first view and second view suggest that the threads of program A have the greatest potential for optimisation. The representation in the last view is a bit less dramatic for the "Executing" metric, but clearly indicates that there is an issue with data locality for the threads of program A. Program B also has a few outliers that would be worth investigating.*"

RQ 3.2: Does the visualisation help to reduce cognitive load of the data locality identification task? While we did not observe any effect of measured variables on the perceived difficulty (ie: on answers regarding the question *How challenging was the problem?*), we received a significant amount of feedback which suggests that the visualisation effectively supports programmers and reduced the cognitive load, which we believe to be higher for experts as they have significantly more complex mental models of data locality effects on performance.

Veteran: "*[It's suprising] how easy it is to see where the issues are.*"

We have observed that many experts were trying to calculate the amount of data being loaded in/out of memory and comparing this with the cache capacity and the

target architecture, some even did more accurate calculations on paper.

Senior: *"The structure of the data in Col A uses 350 more bytes per Account than that of program B (less the length of pointers etc). So for each account there is much more data to be loaded and processed for Program A. This accounts for the longer run time and the greater interaction with RAM."*

Some of the participants attempted the estimation in more complex situations with several loops.

Junior: *"Program B can keep "first in row" element of outer loop in cache (5000*4B = 20kB ¡ cache) while inner loop causes less caches misses because one of them loads many consecutive elements (used for multiples consecutive inner iterations ... 64B/4B = 16 iterations per cache miss ?)."*

RQ 3.3: Do the potential benefits of a visualisation depend on the level of experience (ie: novice vs advanced)? As depicted in Figure 7.17, the self-assessed confidence in the answers of the participants', classified as **Junior and Senior**, was significantly improved when the they used **Data PAL** . In contrast, the confidence of veteran programmers with 10+ years of experience didn't change significantly. Interestingly enough, while veteran programmers were confident about their answers without the visualisation, their correctness was not significantly higher (but higher nonetheless) than the senior programmers with 5-10 years of experience and they also performed quite poorly without the visualisation at their disposal. While a number of different reasons can be postulated for this (including rapid changes to computer architecture which are difficult to keep up with, and underestimations of the scale of the "memory wall"), it emphasises the importance of tool support.

A think aloud process helped us to gain some understanding on how novices look at data locality. In fact, novices were even more surprised than seniors or veteran programmers when it comes to the impact of data locality. In the case quoted below, a participant could not tell the difference between the performance of various implementations by simply looking at the source code (Figures 7.11 and 7.12), while the same problem was easily solved without the visualisation by other, more experienced participants.

Junior: *"I can't tell the difference. They are almost the same except for B using a data*

structure to store information."

It is interesting to notice that the same programmer when presented with the visualisation was able to correctly identify poor data locality performance in programs, while still having no clear idea of how and why data locality has a such major impact on the performance.

Junior: "*The data locality of B is much better than A and for some reason this greatly improves performance.*"

In contrast, experienced programmers leveraged the visualisation and were able to explain what was happening in the programs in extensive detail.

Veteran: "*The difference between the two programs is in the way the data is laid out in memory. In A there is a single structure containing id, name, address and reputation. In B there are two structures; the first containing the name and address, the second containing the id, the reputation and a pointer to the name and address. Accordingly, when executing the loop, program A makes accesses at approximately (depends on packing of structures) byte addresses B, B+362, B+724, whereas program B makes accesses at B, B+20, B+40, ... So, B will make 6 or 8 accesses per cache-line fetched, while A will make 2. The impact of this will be that B will run slower.*"

All the quotes above concern the pair of programs "Account Update", with its performance depicted in the Figure 7.10 and the source code can be seen in Figures 7.11 and 7.12.

7.4.1 LIMITATIONS

The study sample included a small number of senior programmers (the middle category), with greater numbers of veterans and novices. While the performance of this group appears to lie between the other two (which would be expected), the numbers involved are too small to draw any firm conclusions. Only one of the study participants was female unfortunately. While there is no hypothesis regarding gender effects, a more balanced sample would be desirable. As discussed in the experimental design section, there are many tradeoffs in choosing comparison points, while it was decided that comparing against source code only was the most insightful for an initial study,

future work should look at comparing different forms of tool support. Another difficulty with experimentation on any new software tool is the potential influence of training time, particularly with regard to settle-down time for user task performance. While we did not look at performance time for this experiment, future work should also investigate performance time, having made appropriate provision for a learning period within the experimental protocol. In order to make the experiments tractable in terms of time and complexity, only small programs were used. Sophisticated compiler optimisations are also a potential confounding factor in experimentation, further complicating the task facing the programmer. In future work, more ecologically valid experimentation with real programs and real performance problems would be better, although this requires a different approach to experimental design as quantitative data analysis becomes even more difficult to perform.

7.5 CONCLUDING REMARKS

In this chapter we have presented a controlled experiment exploring a software performance visualisation approach with the goal of identifying the presence of data locality issues. Participants were distributed across industry and academia with varying levels of experience.

The visualisation is based on data from a range of hardware performance counters which are processed to make it more easily accessible and represented in a number of ways, including a summary overview, timeline, and (multivariate) thread views. The programmers had no difficulty in understanding these visualisations.

In terms of validation the study shows that this approach leads to a significant improvement in the successful identification of programs and threads exhibiting poor data locality symptoms. While this was unsurprising, it was interesting that this applied across all levels of experience. It was also interesting that the problem of data locality seems to be underestimated even by veteran programmers in terms of its impact on the performance of the program, while some novice programmers were not able to diagnose poor data locality at all without the visualisation.

Veteran programmers were as confident without the visualisation as with it; while it is possible that this would change in a more realistic setting (ie: where programs to

evaluate are larger than just a few lines of code). In contrast, programmers with less than 10 years of experience received a confidence boost in their own diagnosis when using the visualisation.

During the design process of the **Data PAL** , we employed the general framework aimed to support designers of parallel performance tools, more specifically:

- we employed the taxonomy and built a visualisation that supports the diagnosis process of several related problems;
- we used our observational model in order to identify appropriate candidates (counters, events and metrics) for visualisation;
- we used the design advice and provided support for active experimentation and scoping.

Overall, as the evaluation suggests, programmers are able to identify the presence of the performance problems, even in extreme cases where participants did not have previous exposure to the performance problem.

CHAPTER 8 CONCLUSIONS AND FUTURE WORK

In this thesis we have presented a general framework aimed to support designers of parallel performance analysis tools. We have considered many different aspects related to it - from understanding field practices to effective visualisation metaphors and ways of collecting and analysing relevant hardware performance data. The framework we have constructed consists of several major components including: general advice for tool developers, a parallel performance problem taxonomy, an observational model for "data-to-problem" mapping, a deeper analysis of data locality problem identification and a visualisation tool which we have used to evaluate the effectiveness of the approach.

The thesis illustrates the potential of performance data visualisations to increase awareness of the importance and impact of performance problems in parallel software, and is part of a wider research agenda to provide better support for the task of parallel programming in the multi-core era. By conducting a series of interviews and experiments with parallel programmers we have collected a series of insights and models that have been tested by constructing and evaluating the visualisation for data locality problem identification.

First, the **fieldwork process**, its **analysis** and the **implications for design** were presented in the Chapter 4. We carried out the fieldwork study to better understand how developers approach parallel programming, along with the issues that they are trying to address, and how software performance analysis systems could help them in their work. As a result, we have identified some challenges in parallel programming.

Most notably, we have identified the importance of the role of orchestration models in parallel software development, along with the way in which correctness and performance issues are inter-linked in parallel programming. We also noted that the probe effect influences programmers' performance analysis and debugging behaviour

and the issues surrounding the complexity of the parallel programming task and environment. While some of the issues identified apply to more traditional software development, in all cases additional dimensions are introduced by parallel programming.

This qualitative study helped the researchers to inform the discussion on potential tool support for developers, and particularly the design of performance analysis tools. This study served as a basis for the modelling, further analysis and the design of the visualisation tool.

However, additional ongoing effort is required to identify issues, emerging practices and design opportunities for support of parallel programmers. Future work is required to extend the set of implications for design presented in this thesis to a more actionable advice for practitioners and tool developers.

Second, in Chapter 5 we have presented the **analysis of the problem space** consisting of a taxonomy of parallel performance problems, grouped into a number of broad, interrelated themes. The taxonomy was used as the foundation of the **observational model** which focuses primarily on concrete problems that have potential to be related to easily-collectable data, rather than more abstract problems relating to the software architecture or overall design. The model was then validated by 10 domain experts where we were able to identify areas with high levels of agreement, which, when combined with data on relative frequency of occurrence, provides promising directions for initial tool support.

Our results indicate a significant agreement among experts with regard to many of the parallel performance problems identified in our taxonomy, particularly on how various problems can be diagnosed. The study has also identified some contentious and exotic issues. Resolving these areas of disagreement might not involve finding the "right" answer but rather a more nuanced analysis of the problem. The observation might be context-dependent or require simultaneous consideration of multiple pieces of data. Moreover, our model was based on the data we are able to collect, hence, improving and re-evaluating the model with different observational data might lead to significant improvement among expert agreement and potential identification of more nuanced parallel performance problems.

While the observational model was designed with the primary intent to inform the

tool builders, it might also provide a useful starting point for educators, as it has been reported to us that students are often at a loss to understand parallel performance of real programs, partly because they are unaware of the kinds of problems that might exist. We hope that our taxonomy will be a useful starting point for future research on understanding and diagnosing parallel performance problems.

Future work may extend and consolidate the taxonomy for performance problems in programming in general, as parallel and traditional programming are closely interwoven. Additionally, the observational model should be extended based on a robust taxonomy and consider the difference of the impact per architecture such as GPGPU or HPC as well as per type of the application. Performance problems present in a database-type applications might be considerably different from the types of performance problems that occur within a real-time game engine. Moreover, further visualisation techniques can be applied and the need for good performance analysis tools is much greater than ever before.

Third, a deep analysis of **data locality problem** and the **data collection process** were presented in the Chapter 6. We have examined the data locality problem, both its architectural and its algorithmic causes, along with some of the measurable counters and events that can be collected in order to help with the diagnosis of poor data locality in parallel software.

Future collaboration between research and hardware manufacturers is required in the effort to standardise hardware performance counters and their extraction, as they have proven extremely useful for identifying data locality issues. Moreover, some of the data, that is useful for identification, can not be easily collected (e.g. DRAM paging) and requires additional integration from hardware manufacturers.

We expect that future research will deepen the analysis of both problems and the information useful for the identification process; for example, more robust probabilistic metrics can be created to assess the performance impact of particular observable data.

Lastly, in Chapter 7 we have presented an **interactive visualisation tool** along with its **evaluation** aimed at exploring a software performance visualisation approach with the goal of identifying the presence of data locality issues. For the experiment we recruited participants across industry and academia with varying levels of experi-

ence. While we have explored only a single category within our taxonomy, further research should explore more of the solution space by providing tools for other problems within the proposed taxonomy.

In terms of validation, the study shows that this approach can lead to a significant improvement in the successful identification of programs and threads exhibiting poor data locality symptoms. While this was unsurprising, it was interesting that this applied across all levels of experience. It was also interesting that the problem of data locality seems to be underestimated even by veteran programmers in terms of its impact on the performance of the program, while some novice programmers were not able to diagnose poor data locality at all without the visualisation.

Future work is required to design, implement and evaluate the visualisation tools for other problems within the taxonomy we have presented. Such problem-centric parallel performance problem identification tools need to be evaluated and compared to more common and exploratory types of tools.

Overall, this thesis can be seen as a starting point for further HCI research on parallel performance problem identification. While it can and, arguably, should be extended throughout, we have successfully applied our own design advice, taxonomy, models and constructed a visualisation tool that has proven not only to be useful, but also have received extremely positive feedback from the community of practitioners.

BIBLIOGRAPHY

[1] The MSDN common patterns for poorly-behaved multithreaded applications.

[2] Intel threading building blocks design patterns, document number 323512-003us. Tech. rep., Intel Corporation, September 2010.

[3] AMD® CodeAnalyst Performance Analyzer. *http://developer.amd.com/tools-and-sdks/archive/amd-codeanalyst-performance-analyzer/* (2014).

[4] Intel 64 and ia-32 architectures optimization reference manual, order no. 248966-030. Tech. rep., Intel Corporation, September 2014.

[5] Intel® VTune Amplifier XE. *https://software.intel.com/en-us/intel-vtune-amplifier-xe* (2014).

[6] A. LAKSBERG, H. SUTTER, A. ROBISON, S. M. A C ++ Library Solution to Parallelism. Tech. rep., INCITS, InterNational Committee for Information Technology Standards, 2012.

[7] ADHIANTO, L., BANERJEE, S., FAGAN, M., KRENTEL, M., MARIN, G., AND TALLENT, N. R. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Context* (2010), 1–7.

[8] AKHTER, S., AND ROBERTS, J. *Multi-core programming: : Increasing Performance through Software Multi-threading*, vol. 33. Intel press Hillsboro, 2006.

[9] AMDAHL, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference* (New York, NY, USA, 1967), AFIPS '67 (Spring), ACM, pp. 483–485.

[10] ANDERSON, T. The performance of spin lock alternatives for shared-money multiprocessors. *Parallel and Distributed Systems, IEEE Transactions on 1*, 1 (Jan 1990), 6–16.

[11] ANNAVARAM, M., RAKVIC, R., POLITO, M., BOUGUET, J.-Y., HANKINS, R., AND DAVIES, B. The Fuzzy Correlation between Code and Performance Predictability. *37th International Symposium on Microarchitecture (MICRO-37'04)* (2004), 93–104.

[12] ATACHIANTS, R., GREGG, D., JARVIS, K., AND DOHERTY, G. Design considerations for parallel performance tools. In *CHI '14 Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2014), pp. 2501–2510.

[13] BALL, L., AND ORMEROD, T. Structured and opportunistic processing in design: A critical discussion. *International Journal of Human-Computer Studies* (1995).

[14] BEGEL, A., AND ZIMMERMANN, T. Analyze This ! 145 Questions for Data Scientists in Software Engineering. In *ICSE '14* (2014), pp. 12–23.

[15] BEIZER, B. *Software Testing Techniques (2Nd Ed.)*. Van Nostrand Reinhold Co., 1990.

[16] BEN-ARI, M. *Principles of Concurrent Programming*. Prentice Hall Professional Technical Reference, 1982.

[17] BISHOP, M., DILGER, M., ET AL. Checking for race conditions in file accesses. *Computing systems 2*, 2 (1996), 131–152.

[18] BLASGEN, M., GRAY, J., MITOMA, M., AND PRICE, T. The convoy phenomenon. *SIGOPS Oper. Syst. Rev. 13*, 2 (Apr. 1979), 20–25.

[19] BOLOSKY, W. J., SCOTT, M. L., FITZGERALD, R. P., FOWLER, R. J., AND COX, A. L. Numa policies and their relation to memory architecture. *SIGARCH Comput. Archit. News 19*, 2 (Apr. 1991), 212–221.

[20] BONAR, J., AND LIFFICK, B. A visual programming language for novices. Tech. rep., 1987.

[21] BONAR, J., AND SOLOWAY, E. Preprogramming Knowledge: A Major Source of Misconceptions in Novice Programmers. *Human-Computer Interaction 1*, 2 (1985), 133–161.

[22] BOULAY, B., O'SHEA, T., AND MONK, J. The black box inside the glass box: Presenting computing concepts to novices. *International Journal of Human-Computer Studies 51*, 2 (1999), 265–277.

[23] BRESHEARS, C. *The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications*. O'Reilly Media, Inc., 2009.

[24] BURTSCHER, M., KIM, B.-D., DIAMOND, J., MCCALPIN, J., KOESTERKE, L., AND BROWNE, J. Perfexpert: An easy-to-use performance diagnosis tool for hpc applications. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (Washington, DC, USA, 2010), SC '10, IEEE Computer Society, pp. 1–11.

[25] CAMPBELL, D. K. G. Towards the Classification of Algorithmic Skeletons. Tech. rep., 1996.

[26] CANTRILL, B., SHAPIRO, M., AND LEVENTHAL, A. Dynamic Instrumentation of Production Systems. *USENIX Annual Technical ...* (2004).

[27] CARD, S. K., MACKINLAY, J. D., AND SHNEIDERMAN, B. *Readings in information visualization: using vision to think*. Morgan Kaufmann Publishers, 1999.

[28] CAROLINA, N., MURPHY-HILL, E., ZIMMERMANN, T., AND NAGAPPAN, N. Cowboys , Ankle Sprains , and Keepers of Quality : How Is Video Game Development Different from Software Development ? In *ICSE '14* (2014), pp. 1–11.

[29] CARR, S., MCKINLEY, K. S., AND TSENG, C.-W. Compiler optimizations for improving data locality. *SIGOPS Oper. Syst. Rev. 28*, 5 (Nov. 1994), 252–262.

[30] CASAVANT, T. Tools and Methods for Visualization of Parallel Systems and Computations Guest Editor's Introduction, June 1993.

[31] CHANDRA, R., DAGUM, L., KOHR, D., MAYDAN, D., MCDONALD, J., AND MENON, R. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[32] CHARTERS, S. M., THOMAS, N., AND MUNRO, M. The end of the line for software visualisation? In *In Proceedings of the 2nd Workshop on Visualizing Software for Analysis and Understanding. Society Press* (2003), Citeseer, pp. 110–112.

[33] CHEN, C. *Information Visualization: Beyond the Horizon*, 2nd ed. ed. Springer-Verlag, 2006.

[34] CLEVELAND, W. S., AND MCGILL, R. Graphical Perception: Theory, Experimentation, and Application to the Development of Graphical Methods. *Journal of the American Statistical Association 79*, 387 (Sept. 1984), 531.

[35] COLE, M. I. *Algorithmic Skeletons : Structured Management of Parallel Computation Table of Contents*. MIT Press Cambridge, MA, USA, 1991.

[36] CONSTANTINOU, T., SAZEIDES, Y., MICHAUD, P., FETIS, D., AND SEZNEC, A. Performance implications of single thread migration on a chip multi-core. *SIGARCH Comput. Archit. News 33*, 4 (Nov. 2005), 80–91.

[37] CRESWELL, J. W., AND CLARK, V. L. P. Designing and conducting mixed methods research.

[38] DE PAUW, W., JENSEN, E., MITCHELL, N., SEVITSKY, G., VLISSIDES, J., AND YANG, J. Visualizing the execution of java programs. In *Software Visualization*. Springer, 2002, pp. 151–162.

[39] DHRUBAJYOTI GOSWAMI, AJIT SINGH, B. R. P. Architectural Skeletons: The Re-Usable Building-Blocks for Parallel Applications. In *Proc. 1999 International Conference on Parallel and Distributed Processing Techniques and Applications* (1999), pp. 1250–1256.

[40] DONALDSON, D. D., HOWARD, M. N., ORBITS, D. A., PARCHEM, J. M., ROBINSON, D. M., AND WILLIAMS, D. Cache coherency protocol for multi processor computer system, Mar. 22 1994. US Patent 5,297,269.

[41] DREPPER, U. What Every Programmer Should Know About Memory. Tech. rep., Red Hat, 2007.

[42] DRISCOLL, D. L., APPIAH-YEBOAH, A., SALIB, P., AND RUPERT, D. J. Merging qualitative and quantitative data in mixed methods research: How to and why not. *Ecological and Environmental Anthropology (University of Georgia)* (2007), 18.

[43] DRONGOWSKI, J. G. P. An Exploratory Study of Computer Program Debugging. *Human Factors 16* (1974), 258–277.

[44] DRONGOWSKI, P. J., TEAM, A. C., AND CENTER, B. D. An introduction to analysis and optimization with amd codeanalyst performance analyzer. *Advanced Micro Devices, Inc* (2008).

[45] ECCLES, R., AND STACEY, D. Understanding the Parallel Programmer. *20th International Symposium on High-Performance Computing* (2006), 12–12.

[46] ENGLER, D., AND ASHCRAFT, K. Racerx: Effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 237–252.

[47] FAHRINGER, T. *Automatic Performance Prediction of Parallel Programs*, 1st ed. Springer Publishing Company, Incorporated, 2011.

[48] FAISAL, S., CAIRNS, P., AND BLANDFORD, A. Challenges of evaluating the information visualisation experience. In *Proceedings of the 21st British HCI Group Annual Conference on People and Computers: HCI...But Not As We Know It - Volume 2* (Swinton, UK, UK, 2007), BCS-HCI '07, British Computer Society, pp. 167–170.

[49] FIX, V., WIEDENBECK, S., AND SCHOLTZ, J. Mental representations of programs by novices and experts. In *Proc. INTERACT '93/ACM CHI '93* (1993), pp. 74–79.

[50] FOTHERINGHAM, J. Dynamic storage allocation in the atlas computer, including an automatic use of a backing store. *Commun. ACM 4*, 10 (Oct. 1961), 435–436.

[51] FRIENDLY, M. A Brief History of Data Visualization. In *Handbook of Computational Statistics*. Springer, 2008, ch. Chapter II, pp. 15–56.

[52] FRITZ, T., AND MURPHY, G. Using information fragments to answer the questions developers ask. *Proc. ACM/IEEE ICSE 2010 1* (2010), 175.

[53] GAIT, J. A probe effect in concurrent programs. *Software, practice & experience 16*, 3 (1986), 225–233.

[54] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[55] GHANAM, Y., AND CARPENDALE, S. A survey paper on software architecture visualization," technical report, 2008.

[56] GOLDENSON, D., AND WANG, B. Use of Structure Editing Tools by Novice Programmers. In *Empirical Studies of Programming: Fourth Work* (1991), pp. 99–120.

[57] GOTO, K., AND GEIJN, R. A. V. D. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw. 34*, 3 (May 2008), 12:1–12:25.

[58] GREEN, T. R. G., AND PETRE, M. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages and Computing 7*, 2 (1996), 131–174.

[59] GREGG, B. *Systems Performance: Enterprise and the Cloud*, 1st ed. Prentice Hall Press, Upper Saddle River, NJ, USA, 2013.

[60] GUGERTY, L., AND OLSON, G. Debugging by skilled and novice programmers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 1986), CHI '86, ACM, pp. 171–174.

[61] GUPTA, A., AND KUMAR, V. Scalability of parallel algorithms for matrix multiplication. In *in Proc. of Int. Conf. on Parallel Processing* (1991), pp. 115–123.

[62] HAEBERLEN, A., AND ELPHINSTONE, K. User-level management of kernel memory. In *Proceedings of the Eighth Asia-Pacific Computer Systems Architecture Conference (ACSAC'03)* (Sep 2003).

[63] HAN, S., DANG, Y., GE, S., AND ZHANG, D. Performance Debugging in the Large via Mining Millions of Stack Traces. In *ICSE '12* (2012), pp. 176–186.

[64] HANNAY, J., MACLEOD, C., SINGER, J., LANGTANGEN, H., PFAHL, D., AND WILSON, G. How do scientists develop and use scientific software? *2009 ICSE Workshop on Software Engineering for Computational Science and Engineering* (May 2009), 1–8.

[65] HEATH, M. Visualizing the performance of parallel programs. *IEEE Software* (1991), 29–39.

[66] HEATH, M., A.D. MALONY, AND ROVER, D. Parallel performance visualization: from practice to theory. *IEEE Parallel & Distributed Technology: Systems & Applications 3*, 4 (1995), 44–60.

[67] HEATH, M., AND MALONY, A. The visual display of parallel performance data. *IEEE Computer* (1995).

[68] HEER, J., KONG, N., AND AGRAWALA, M. Sizing the horizon: The effects of chart size and layering on the graphical perception of time series visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2009), CHI '09, ACM, pp. 1303–1312.

[69] HEIRMAN, W., CARLSON, T., VAN CRAEYNEST, K., HUR, I., JALEEL, A., AND EECKHOUT, L. Undersubscribed threading on clustered cache architectures. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on* (Feb 2014), pp. 678–689.

[70] HENNESSY, J. L., AND PATTERSON, D. A. *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011.

[71] HENNESSY, J. L., AND PATTERSON, D. A. *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011.

[72] HERB SUTTER. Understanding Parallel Performance. *Dr. Dobb's Journal* (2008).

[73] HERLIHY, M., AND SHAVIT, N. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[74] HOCHSTEIN, L., AND CARVER, J. Parallel programmer productivity: A case study of novice parallel programmers. *High Performance Networking and Computing* (2005), 1–9.

[75] HUANG, Y., CUI, Z., CHEN, L., AND ZHANG, W. HaLock: Hardware-assisted lock contention detection in multithreaded applications. In *PACT '12: Proceedings of the 21st international conference on Parallel architectures and compilation techniques* (2012).

[76] HUCK, K. A., AND MALONY, A. D. Perfexplorer: A performance data mining framework for large-scale parallel computing. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing* (Washington, DC, USA, 2005), SC '05, IEEE Computer Society, pp. 41–.

[77] IANCU, C., HOFMEYR, S., BLAGOJEVIC, F., AND ZHENG, Y. Oversubscription on multicore processors. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on* (April 2010), pp. 1–11.

[78] INSELBERG, A. Multidimensional detective. In *Information Visualization, 1997. Proceedings., IEEE Symposium on* (Oct 1997), pp. 100–107.

[79] IPEK, E., DE SUPINSKI, B., AND SCHULZ, M. An approach to performance prediction for parallel applications. *Euro-Par 2005 Parallel* (2005), 196–205.

[80] JOHNSON, R. B., AND ONWUEGBUZIE, A. J. Mixed methods research: A research paradigm whose time has come. *Educational researcher 33*, 7 (2004), 14–26.

[81] JOVIC, M., ADAMOLI, A., AND HAUSWIRTH, M. Catch Me If You Can : Performance Bug Detection in the Wild. In *OOPSLA* (2011), pp. 155–170.

[82] KANDIRAJU, G. B., AND SIVASUBRAMANIAM, A. Characterizing the d-TLB behavior of SPEC CPU2000 benchmarks. *SIGMETRICS Perform. Eval. Rev. 30*, 1 (June 2002), 129–139.

[83] KEIM, D., KHOLHAMMER, J., ELLIS, G., AND FLORIAN MANSMANN. *Mastering the Information Age: Solving Problems with Visual Analytics.* Eurographics Association, 2010.

[84] KENNEDY, K., AND MCKINLEY, K. S. Optimizing for parallelism and data locality. In *ACM International Conference on Supercomputing 25th Anniversary Volume* (New York, NY, USA, 2014), ACM, pp. 151–162.

[85] KHAN, T., BARTHEL, H., EBERT, A., AND LIGGESMEYER, P. Visualization and evolution of software architectures.

[86] KIM, S., CHANDRA, D., AND SOLIHIN, Y. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques* (Washington, DC, USA, 2004), PACT '04, IEEE Computer Society, pp. 111–122.

[87] KO, A. J., MYERS, B. A., COBLENZ, M. J., AND AUNG, H. H. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Trans. Softw. Eng. 32*, 12 (Dec. 2006), 971–987.

[88] KOENEMANN, J., AND ROBERTSON, S. Expert problem solving strategies for program comprehension. *Proc ACM CHI '91* (1991), 125–130.

[89] KOSCHKE, R. Software visualization in software maintenance, reverse engineering, and re-engineering: A research survey. *Journal of Software Maintenance 15*, 2 (Mar. 2003), 87–109.

[90] KRAEMER, E., AND STASKO, J. The visualization of parallel systems: An overview.

[91] LANDIS, J. R., AND KOCH, G. G. The measurement of observer agreement for categorical data. *Biometrics 33* (1977), 159–174.

[92] LANGELIER, G., SAHRAOUI, H., AND POULIN, P. Exploring the evolution of software quality with animated visualization. In *Proceedings of the 2008 IEEE Symposium on Visual Languages and Human-Centric Computing* (Washington, DC, USA, 2008), VLHCC '08, IEEE Computer Society, pp. 13–20.

[93] LATOZA, T., AND MYERS, B. Hard to answer questions about code. In *"Second Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU'2010) at SPLASH/Onward!"* (2010).

[94] LAWRANCE, J., BOGART, C., BURNETT, M., AND BELLAMY, R. How people debug, revisited: An information foraging theory perspective. *IEEE Transactions on Software Engineering 39*, 2 (2009), 197–215.

[95] LEDOUX, C., AND JR, D. P. Saving Traces for ADA Debugging. *ACM SIGAda Ada Letters* (1985).

[96] LEE, E. A. Problem with Threads. *Computer*, May (2006), 33–42.

[97] LEE, J., WU, H., RAVICHANDRAN, M., AND CLARK, N. Thread tailor: Dynamically weaving threads together for efficient, adaptive parallel applications. In *Proceedings of the 37th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2010), ISCA '10, ACM, pp. 270–279.

[98] LEWIS, C., LIN, Z., SADOWSKI, C., ZHU, X., OU, R., AND JR, E. J. W. Does Bug Prediction Support Human Developers ? Findings from a Google Case Study. In *ICSE '13 Proceedings of the 2013 International Conference on Software Engineering* (2013), pp. 372–381.

[99] LIINSKY, N., AND STEELE, J. *Data Visualizations*. O'Reilly, 2011.

[100] LINDQUIST, E. Background paper Surveying the world of visualization.

[101] LIU, J., NICOL, D., AND PREMORE, B. Performance Prediction of a Parallel Simulator. *Distributed Simulation,* (1999).

[102] LUFF, M. Empirically investigating parallel programming paradigms: A null result. *Usability of Programming Languages and Tools*, October (2009).

[103] MAHAPATRA, N. R., AND VENKATRAO, B. The processor-memory bottleneck: Problems and solutions. *Crossroads 5*, 3es (Apr. 1999).

[104] MAK, J., AND MYCROFT, A. Limits of parallelism using dynamic dependency graphs. In *Proceedings of the Seventh International Workshop on Dynamic Analysis* (New York, NY, USA, 2009), WODA '09, ACM, pp. 42–48.

[105] MARCUS, A., FENG, L., AND MALETIC, J. I. 3d representations for software visualization. In *Proceedings of the 2003 ACM Symposium on Software Visualization* (New York, NY, USA, 2003), SoftVis '03, ACM, pp. 27–ff.

[106] MATTSON, T., AND WRINN, M. Parallel programming: can we PLEASE get it right this time? *Proceedings of the 45th annual Design Automation ...* (2008), 7–11.

[107] MAYER, R. The Psychology of How Novices Learn Computer Programming. *ACM Computing Surveys 13*, 1 (Jan. 1981), 121–141.

[108] MAZZA, R. *Introduction to Information Visualization*. Springer-Verlag, 2009.

[109] MCCANDLESS, D. *Information is Beautiful*, 2nd ed. ed. Collins, 2012.

[110] MCCAULEY, R., FITZGERALD, S., LEWANDOWSKI, G., MURPHY, L., SIMON, B., THOMAS, L., AND ZANDER, C. Debugging: a review of the literature from an educational perspective. *Computer Science Education 18*, 2 (June 2008), 67–92.

[111] MCCORMICK, B. H. Visualization in scientific computing. *ACM SIGBIO Newsletter 6*, November 1987 (1991), 15–21.

[112] MCKENNEY, P., GUPTA, M., MICHAEL, M., HOWARD, P., TRIPLETT, J., AND WALPOLE, J. Is parallel programming hard, and if so, why? *Control* (2002).

[113] MCVOY, L., GRAPHICS, S., STAELIN, C., AND LABORATORIES, H.-P. lmbench : Portable Tools for Performance Analysis lmbench : Portable tools for performance analysis.

[114] MILLER, B., CALLAGHAN, M., CARGILLE, J., HOLLINGSWORTH, J., IRVIN, R., KARAVANIC, K., KUNCHITHAPADAM, K., AND NEWHALL, T. The Paradyn parallel performance measurement tool. *Computer 28*, 11 (1995), 37–46.

[115] MULLER, M., AND KOGAN, S. Grounded theory method in hci and cscw. *Cambridge: IBM Center for Social ...* (2010), 1–46.

[116] MURRAY, S. *Interactive Data Visualization*. O'Reilly, 2013.

[117] MYERS, R. Funny, it worked last time: Event tracing for windows (etw), 2005.

[118] MYTKOWICZ, T., DIWAN, A., HAUSWIRTH, M., AND SWEENEY, P. F. Evaluating the accuracy of java profilers. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2010), PLDI '10, ACM, pp. 187–197.

[119] NAVARRO, A., AND ZAPATA, E. An automatic iteration/data distribution method based on access descriptors for dsmm. In *Languages and Compilers for Parallel Computing*, L. Carter and J. Ferrante, Eds., vol. 1863 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2000, pp. 133–148.

[120] NETZER, R. H. B., AND MILLER, B. P. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst. 1*, 1 (Mar. 1992), 74–88.

[121] NISTOR, A., SONG, L., MARINOV, D., AND LU, S. Toddler : Detecting Performance Problems via Similar Memory-Access Patterns. In *ICSE '13 Proceedings of the 2013 International Conference on Software Engineering* (2013), pp. 562–571.

[122] NIU, N., MAHMOUD, A., CHEN, Z., AND BRADSHAW, G. Departures from optimality: understanding human analyst's information foraging in assisted requirements tracing. *Proc. ICSE '13* (2013), 572–581.

[123] NORMAN, D. A. *The design of everyday things*. 2002.

[124] ODAIRA, R., CASTANOS, J. G., AND TOMARI, H. Eliminating global interpreter locks in Ruby through hardware transactional memory. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2014), PPoPP '14, ACM, pp. 131–142.

[125] PANAS, T., EPPERLY, T., QUINLAN, D., SAEBJORNSEN, A., AND VUDUC, R. Communicating software architecture using a unified single-view visualization. In *Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on* (July 2007), pp. 217–228.

[126] PANCAKE, C. Applying human factors to the design of performance tools. 44–60.

[127] PANCAKE, C., AND UTTER, S. Models for visualization in parallel debuggers. In *Proc. ACM/IEEE conference on Supercomputing* (1989), pp. 627–636.

[128] PANE, J., AND MYERS, B. Usability issues in the design of novice programming systems. Tech. Rep. August, 1996.

[129] PAPAMARCOS, M. S., AND PATEL, J. H. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture* (New York, NY, USA, 1984), ISCA '84, ACM, pp. 348–354.

[130] PARK, I., AND RAGHURAMAN, M. K. Server diagnosis using request tracking. In *1st Workshop on the Design of Self-Managing Systems, held in conjunction with DSN 2003* (2003).

[131] PERKINS, D., AND MARTIN, F. Fragile knowledge and neglected strategies in novice programmers. Tech. rep., 1986.

[132] PIORKOWSKI, D., FLEMING, S., KWAN, I., BURNETT, M., SCAFFIDI, C., BELLAMY, R., AND JORDAHL, J. The whats and hows of programmers' foraging diets. *Proc. ACM CHI 2013* (2013), 3063–3072.

[133] PIROLLI, P., AND CARD, S. Information foraging in information access environments. *Proc. CHI '95* (1995), 51–58.

[134] PRABHU, P., ZHANG, Y., GHOSH, S., AUGUST, D., HUANG, J., BEARD, S., KIM, H., OH, T., JABLIN, T., JOHNSON, N., ZOUFALY, M., RAMAN, A., LIU, F., AND WALKER, D. A survey of the practice of computational science. *State of the Practice Reports on - SC '11* (2011), 1.

[135] QUANTE, J. Do dynamic object process graphs support program understanding? - a controlled experiment. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on* (June 2008), pp. 73–82.

[136] RAJWAR, R., AND GOODMAN, J. R. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture* (Washington, DC, USA, 2001), MICRO 34, IEEE Computer Society, pp. 294–305.

[137] REINDERS, J. *VTune performance analyzer essentials*. Intel Press, 2005.

[138] REINDERS, J. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*, 1st ed. O'Reilly Media, Inc, 2007.

[139] RENDGEN, S., AND WIEDEMANN, J. *Information Graphics*. Taschen, 2012.

[140] RICHARD WETTEL, M. L., AND ROBBES, R. Empirical validation of codecity: A controlled experiment. Tech. Rep. 2010/05, University of Lugano, June 2010.

[141] ROMAN, G.-C., AND COX, K. C. Program visualization: The art of mapping programs to pictures. In *Proceedings of the 14th International Conference on Software Engineering* (New York, NY, USA, 1992), ICSE '92, ACM, pp. 412–420.

[142] RUSSELL, K., AND DETLEFS, D. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications* (New York, NY, USA, 2006), OOPSLA '06, ACM, pp. 263–272.

[143] RYAN NEWTON, FRANK SCHLIMBACH, MARK HAMPTON, K. K. Capturing and Composing Parallel Patterns with Intel CnC. In *HotPar '10* (2010).

[144] S. FLEMING. *Successful Strategies for Debugging Concurrent Software: An Empirical Investigation*. PhD thesis, 2009.

[145] SADOWSKI, C., AND SHEWMAKER, A. The Last Mile : Parallel Programming and Usability. *FOSER* (2010).

[146] SHENDE, S. S., AND MALONY, A. D. The tau parallel performance system. *International Journal of High Performance Computing Applications 20*, 2 (2006), 287–311.

[147] SIEGMUND, N., KOLESNIKOV, S. S., CHRISTIAN, K., APEL, S., AND SAAKE, G. Predicting Performance via Automated Feature-Interaction Detection. In *ICSE '12* (2012), no. ii, pp. 167–177.

[148] SILLITO, J., MURPHY, G., AND DE VOLDER, K. Questions programmers ask during software evolution tasks. In *SIGSOFT'06/FSE-14: Proceedings of the 13th ACM SIGSOFT and 14th international symposium on Foundations of Software Engineering* (2006).

[149] SIU, S., SIMONE, M. D., GOSWAMI, D., AND SINGH, A. Design Patterns for Parallel Programming. *PDPTA* (1996).

[150] SMITH, D., CYPHER, A., AND SPOHRER, J. KidSim: programming agents without a programming language. *Communications of the ACM 37*, 7 (June 1994), 54–67.

[151] SNYDER, L. Parallel programming and the poker programming environment. Tech. rep., DTIC Document, 1984.

[152] SNYDER, L., AND SOCHA, D. Poker on the cosmic cube: The first retargetable parallel programming language and environment. Tech. rep., DTIC Document, 1986.

[153] SOLOWAY, E., LAMPERT, R., AND LETOVSKY, S. Designing documentation to compensate for delocalized plans. *Communications of the ACM 31*, 11 (1988).

[154] SPENCE, R. *Information Visualization: Design for Interaction (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2007.

[155] STOL, K.-J., AND FITZGERALD, B. Two ' s Company , Three ' s a Crowd : A Case Study of Crowdsourcing Software Development. *ICSE '14* (2014), 187–198.

[156] STOREY, M.-A. D., FRACCHIA, F. D., AND MÜLLER, H. A. Cognitive design elements to support the construction of a mental model during software exploration. *J. Syst. Softw. 44*, 3 (Jan. 1999), 171–185.

[157] STRAUSS, A. L., AND CORBIN, J. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*, 2nd ed. Sage, Thousand Oaks, 1998.

[158] SÜSS, M., AND LEOPOLD, C. Common mistakes in OpenMP and how to avoid them. In *Proceedings of the 2005 and 2006 international conference on OpenMP shared memory parallel programming*. Springer, 2008, pp. 312–323.

[159] TALLENT, N., MELLOR-CRUMMEY, J., AND PORTERFIELD, A. Analyzing lock contention in multithreaded applications. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2010).

[160] TILLEY, S., AND HUANG, S. Documenting software systems with views iii: Towards a task-oriented classification of program visualization techniques. In *Proceedings of the 20th Annual International Conference on Computer Documentation* (New York, NY, USA, 2002), SIGDOC '02, ACM, pp. 226–233.

[161] TORRELLAS, J., LAM, M., AND HENNESSY, J. L. False sharing and spatial locality in multiprocessor caches. *Computers, IEEE Transactions on 43*, 6 (Jun 1994), 651–663.

[162] TRINDER, P. W., K. HAMMOND, H.-W. LOIDL, S. L. P. J. Algorithm + strategy = parallelism. *Journal of Functional Programming 8*, 1 (1998), 23–60.

[163] TUFTE, E. Envisioning information. *Optometry & Vision Science* (1991).

[164] TUFTE, E. *The Visual Display of Quantitative Information*, 2nd ed. ed. Graphics Press, 2001.

[165] VESSEY, I. Expertise in Debugging Computer Programs: An Analysis of the Content of Verbal Protocols. *IEEE Transactions on Systems, Man, and Cybernetics 16*, 5 (Sept. 1986), 621–637.

[166] VIERA, A. J., GARRETT, J. M., ET AL. Understanding interobserver agreement: the kappa statistic. *Fam Med 37*, 5 (2005), 360–363.

[167] WAHEED, A., AND ROVER, D. Performance visualization of parallel programs. In *VIS '93: Proceedings of the 4th conference on Visualization '93* (1993).

[168] WETTEL, R., AND LANZA, M. Codecity: 3d visualization of large-scale software. In *Companion of the 30th International Conference on Software Engineering* (New York, NY, USA, 2008), ICSE Companion '08, ACM, pp. 921–922.

[169] WETTEL, R., LANZA, M., AND ROBBES, R. Software systems as cities: A controlled experiment. In *Proceedings of the 33rd International Conference on Software Engineering* (New York, NY, USA, 2011), ICSE '11, ACM, pp. 551–560.

[170] WHITE, R. W., KULES, B., DRUCKER, S. M., AND SCHRAEFEL, M. Supporting exploratory search. *Communications of the ACM 49*, 4 (Apr. 2006), 36–39.

[171] WOLFE, M., AND BANERJEE, U. Data dependence and its application to parallel processing. *Int. J. Parallel Program. 16*, 2 (Apr. 1987), 137–178.

[172] WULF, W. A., AND MCKEE, S. A. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news 23*, 1 (1995), 20–24.

[173] YANG, L., AND MUELLER, F. Cross-Platform Performance Prediction of Parallel Applications Using Partial Execution. *ACM/IEEE SC 2005 Conference (SC'05)* (2005), 40–40.

[174] YANG, L., AND MUELLER, F. Cross-Platform Performance Prediction of Parallel Applications Using Partial Execution. *ACM/IEEE SC 2005 Conference (SC'05)* (2005), 40–40.

[175] YAU, N. *Data Points*. John Wiley & Sons, Inc., 2013.

[176] YOO, W., LARSON, K., AND BAUGH, L. ADP: automated diagnosis of performance pathologies using hardware events. *SIGMETRICS* (2012), 283–294.

[177] ZAMAN, S., ADAMS, B., AND HASSAN, A. E. A Qualitative Study on Performance Bugs. In *MSR* (2012), pp. 199–208.

[178] ZHANG, Y., KANDEMIR, M., AND YEMLIHA, T. Studying inter-core data reuse in multicores. In *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2011), SIGMETRICS '11, ACM, pp. 25–36.

[179] ZHURAVLEV, S., BLAGODUROV, S., AND FEDOROVA, A. Addressing shared resource contention in multicore processors via scheduling. In *ACM SIGARCH Computer Architecture News* (2010), vol. 38, ACM, pp. 129–142.