This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2016.2519346, IEEE Transactions on Software Engineering

Parallel Performance Problems on Shared-Memory Multicore Systems: Taxonomy and Observation

Roman Atachiants, Gavin Doherty and David Gregg

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 optimizing 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. 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.

1 Introduction

As computers become increasingly involved in every aspect of our lives, the tasks that computers may perform have also become more complex and the volume of data processed by computers has increased enormously. These more complex tasks and increased volumes of data have required the development of ever more computationally demanding applications [1]. For many decades it was possible to meet the corresponding demand for computational capacity by improving single-threaded performance. In particular, increasing transistor density allowed faster clock frequencies by packing more hardware within a smaller area, and more sophisticated architectures that allow more work to be completed within a given time. However, this approach resulted in an unsustainable upward trend in processor energy consumption. Increasing clock frequencies resulted in large increases in energy consumption, which in turn must be dissipated as heat within the small area of a processor [2]. An alternative strategy for increasing processor performance is to put multiple processors (cores) on the same chip . Typical modern personal computers now have between two and eight cores that enable multiple tasks (threads) to be executed simultaneously. Multicore processors are now found in devices from smart-phones and games consoles through to servers and supercomputers [3].

• *All authors are based in Trinity College Dublin*

Going forward, computers are expected to have even more cores. Hardware transistor densities continue to double around every two years, in line with Moore's law. This doubling in transistor density opens the possibility of the number of cores doubling with each new generation [2]. However, exploiting dozens or hundreds of cores to cooperatively solve a single problem is extremely challenging. Some problems, such as web servers and some map-reduce problems are inherently parallel and can exploit many cores. But where large numbers of cores must work interdependently to solve a single problem, good performance can only be achieved with careful attention to performance issues in the parallelization strategy [4].

A programmer seeking to parallelize a program for multicore computing has to overcome challenges including synchronization, non-determinism and orchestration, that a programmer writing an equivalent sequential program would not have to face [5]. Additionally, the very process of parallelizing software may introduce bugs, deadlocks, race conditions and other problems into the program. To further complicate matters, when developing a program it is not necessarily obvious what its parallel performance will be.

Since at least the 1950s, researchers have attempted to improve computer performance by exploiting multiple processors [6]. During that time a great deal of progress has been made, particularly on parallel computer hardware, and to a lesser extent on software. For most of these decades parallel computing has primarily involved large, expensive supercomputers, solving problems that are well-suited to parallelization, and programmed by expert parallel software developers [7]. These supercomputers often have hundreds or thousands of separate processors, and rely on abundant parallelism and relatively simple array-based data structures and control flow in the parallel programs. The multicore era is different in that the challenge is for the average programmer to build parallel applications to solve a wide variety of more general problems, and which execute on a single multicore processor [8].

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 [9]. However, twenty years later, concurrency problems are still very common. According to the UBM TechWeb Survey of 275 software engineers or managers of development teams conducted in October 2011, 32% of participants spend 6+ hours per month finding data races or deadlock conditions and 69% spend 6+ hours per month tuning the performance of their applications[1].

At the point of writing, no formal and well accepted taxonomy of multicore parallel performance problems exists so far as we are aware. Hence, we present an initial taxonomy of multicore parallel performance problems. These are classified into seven interrelated categories, six of which we consider in detail in this paper[2]. Each category contains several specific performance problems that can arise in multicore parallel programs. Note that (as we discuss in more detail in section 3.1) our taxonomy is aimed specifically at problems that arise in general-purpose multicore programming. Thus, we do not consider large-scale parallel supercomputers, distributed memory clusters, or graphics processing units. Instead, we focus on the ordinary developer attempting to exploit multiple cores to speed up their applications. We anticipate that the developer will implement parallelism using threads, locks and shared variables, on a conventional desktop, server, or laptop multicore computer.

There are many possible ways to conceptualize parallel performance problems. Furthermore, certain types of parallel performance problems, such as those arising from the design of the software architecture, are not easily characterized. The model presented here is focused on 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 their own mental representation of the program, and of the problems which might be present in 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. 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 [10]. 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. Both studies are discussed together in Section 6 in order to consider the implications for the analysis of parallel performance data.

1. The parallel programming landscape, UBM TechWeb, 2011
2. One of the categories (I/O) is conceptually distinct and not covered in detail in this paper, for reasons which are discussed later.

2 Related Work

In recent years, there has been a dramatic shift to multiprocessor programming and much work has been done to make development of such programs easier for developers and scientists [11].

Performance problems (or performance bugs) are program features that create significant performance degradation. Although a great deal of research has been devoted to detecting such problems, performance problems regularly persist into production environments, degrade system throughput, waste system resources and affect user experience [12], [13]. Even well-tested major software releases have been affected by significant performance issues [14]. Increasingly, the diagnosis process for performance problems will have to consider the potential role of parallelism, if only to rule it out.

Studies of Programmers. Much work has been carried out within the Software Engineering community with the aim of understanding and modeling the way programmers work and how their workflows can be improved. One of the methodologies applied is to look at how complex strategies can be modelled in a series of simple questions or observations, giving a better understanding of the daily practices of software developers and the architectural choices they face [15], [16], [17].

The practice of software engineering has been examined in various contexts, as the organizational environment can vary drastically. If one considers crowdsourced 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 [18], while video game development, typically conducted by a team of seasoned veterans under one roof presents another set of challenges and pipeline-like organization [19]. Likewise, in multicore software development, both the characteristics of the software being developed, and the development context will have an impact.

Researchers have attempted to understand and model the way programmers debug and navigate through their programs. Much of software developers' time is spent understanding unfamiliar code which they might need to improve, debug or add features to. Studies have shown that developers spend as much as 35 percent of their time navigating the code to locate and understand the parts of the software system relevant to the desired change or maintenance activity [15]. Attempts have been made to model how programmers work from a sensemaking perspective, applying information foraging theory [20] to understand how developers debug [21], [22].

Bug Classifications. It has been argued that within parallel program development, the distinction between performance problems and bugs is much less clear than in traditional software development [23]. However, the vast majority of parallel performance problems would receive only the broadest categorization under existing taxonomies. Within Beizer's taxonomy [24] 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. That is, they are concerned with general symptoms rather than classifications which give some indication of the cause of the problem.

Algorithmic Skeletons. 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" [25], [26], [27], [28]. 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 [29].

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) [30], [31].

Performance Analysis & Prediction. This article presents a model for parallel performance problem diagnosis, with the aim of supporting the design of effective performance analysis tools for parallel programming on multicore systems. 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 [32]. 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 behavior while their individual presence may not [33]. 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, as system complexity increases with size [34], [35], [36].

Most bug/performance prediction algorithms have been developed, tested and verified in an academic setting. However, a recent case study by Lewis et al [37] of a deployment of prediction algorithms, concluded that while many developers are excited about having new tools 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 [13], [14], [38], [39].

Parallel Performance Analysis Tools. An important aspect of tool support for multi-core programming is understanding performance data. Given the volume and complexity of this data, visualization is an important design direction as it leverages capacities and bandwidth of the human visual system to quickly assess and understand large volumes of data. The behaviour of the programs themselves is often complex; designing visualizations of observable data which allow programmers to reason about this complex behaviour is a challenge. The design of effective visualization techniques for parallel programs is still relatively unexplored within parallel programming research and has usability implications [40]. However, the need to form a scientific body of research, develop human-centered models, and target production level applications and their developers has been recognized in the literature [41].

Numerous tools to ease the engineering effort involved in creation, debugging and optimization of parallel program have been created, starting as early as the 1980's with the Poker environment [42], [43], allowing programmers to write and debug the first portable (cross-compiled) parallel programs.

Other tools, such as ParaGraph and ParaDyn have been developed to visualize behaviour of parallel software [44], [45]; most of these tools were 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 regards to tools, existing systems can be seen as addressing two main issues: the 'topology' of software (e.g.: source-code hierarchy, memory layout, etc) and the mapping of such topology into the visualization, and the issue of synchronization. The topology issue requires that spatial relationships in programs be understood. The synchronization issue requires various events occurring within the processor to be correlated [46]. While some existing visualizations are potentially useful, there is a need for analysis of how such tools can aid in the diagnosis of problems [23].

The last decade has seen the development of a number of performance analysis tools, coming from both academia and industry. The rapid growth of distributed computing with the advent of the cloud has increased the need for HPC performance diagnosis tools such as TAU [47] or PerfExplorer [48]. The increased diversity of parallel processors and the number of cores on commodity hardware has led to the creation hardware-specific analysis tools such as Intel VTune Amplifier [49] and AMD CodeAnalyst [50] which leverage hardware performance counters to support "close to the metal" performance diagnosis.

The increased prevalence of parallel hardware has also influenced developers using popular programming environments such Java or .NET, who have sought support for parallelizing and optimizing their systems. To support them and leverage platform-specific constructs such as garbage collectors or just-in-time runtime diagnosis, specialized tools have been created such as Jinsight [51], HProf, XProf, JProfile or YourKit [52].

At the same time, significant efforts within popular operating systems such as Linux or Windows have resulted in better support for parallel hardware and recognized the need for better performance diagnosis of concurrently running processes. One example is the Microsoft Windows Performance Analyzer, based on the event tracing subsystem within the Windows operating system (ETW) [53] which allows visual analysis of every process/thread running on a particular machine.

Finally, Jain [54] deals with performance at a more general level. Although his well-known textbook originated long before the advent of multicore computing, the principles of careful, statistically-sound performance analysis remain valid today.

2.1 Other Domains

Historically the term "parallel computing" has often been associated with supercomputing or high-performance computing, and has mainly been aimed at large-scale scientific applications such as weather forecasting. However, just as much modern multicore computing falls outside this field, there are many other branches of parallel computing that have little to do with traditional supercomputing. For example, Foo et al. [55] propose tools for studying the performance of large-scale enterprise computing systems with large numbers of interacting servers when performance testing might be performed on a variety of different hardware configurations. Chen et al. [56] propose static and dynamic program analysis techniques to detect performance problems in database transactions in large scale enterprise systems. Litoiu and Barna [57] focus on automatic tools for detecting performance problems in parallel web server applications.

The breadth of parallel performance research is almost as broad as the range of parallel computing domains and platforms. All manner of computers from small embedded processors to large distributed enterprise systems exploit parallelism to improve performance in a wide range of problem domains. We focus on one particular area that has become extremely important over the last decade since multicore processors became widely available. The performance of parallel software depends on its interaction with the underlying parallel machine and systems software. Is it possible to define a taxonomy of performance problems that can help the average programmer (who is accustomed to developing sequential software) to identify these problems when attempting to introduce multicore parallelism into their software?

3 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 [23] 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], [58], [59], [60] and books and papers from the research literature on performance of parallel programs [61], [62], [63], [64]. 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 Intels software optimization manual [58]. It specifies five areas for optimization of parallel programs: thread synchronization, bus utilization, memory optimization, front-end optimization and management of shared execution resources [58]. 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 [65], lock contention [66], low-work to synchronization ratio [67], [68], data sharing [2], [58], load balancing [6], TLB locality [69], DRAM memory pages [70], NUMA [58], false sharing [4], and shared resource contention [64].

One of the authors (a domain expert) 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 other 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[3].

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.

3.1 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 network processors have special features to accelerate packet processing.

3. 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

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 [2]. 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[4]. 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 special-purpose network processors [2].

It is important to mention common concurrency problems such as deadlocks and race conditions [71]. 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 [72], [73], 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 [23]. 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.

4. 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.

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 [2]) 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 synchronization 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.

3.2 A Taxonomy of Parallel Performance Problems

The taxonomy is comprised of seven broad categories, presented in the Table 1, with specific and distinct problems within each category. Below we describe each category and the rationale behind the problems included under them.

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 [74] 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

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 NUMA memory shared b/w CPUs 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 1: Taxonomy of parallel performance problems.

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[5]. 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

5. 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.

to exploit the available parallelism [75]. 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 of starting a thread are significant, so a sufficient amount of work must be done by the thread to justify it [5] (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 [76]. 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 [67]. 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[6]. A lock is a small data structure in memory, so there may also be memory caching issues when acquiring a lock.

Lock contention. Lock contention occurs when a thread attempts to acquire a lock but the lock is already held by another thread [78]. 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 [79] 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 implementations 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 [80]. 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

6. 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 [77]. 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.

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 [81]. 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 [82].

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 [83]. 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 [84]. 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 [85]. 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 [6]. 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 [86]. 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 [87].

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 [88]. 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 [89]. 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 [70]. 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 [90]. 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 [91]. 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 [92]. 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" [64].

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 [93]. 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.

3.3 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 figure 6 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.

4 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.

4.1 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 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.

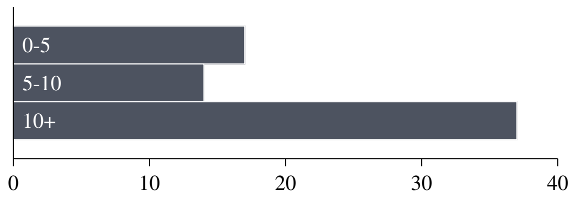

Figure 1: The professional (years of) experience distribution of the developers who participated in the problem frequency/familiarity study.

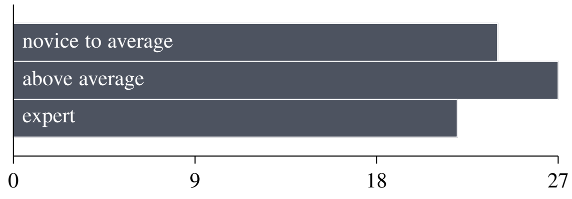

Figure 2: The distribution of self-assessed expertise of the developers who participated in the problem frequency/familiarity study.

4.2 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 1 and 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 (Figure 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 2 will be discussed in more detail in Section 6 together with the study presented in the next 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.

5 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 3.

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 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.

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 contain 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 [49] and TAU [47] 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.

5.1 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 Figure 7 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 [94] 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 puts forward more indications than contra-indications.

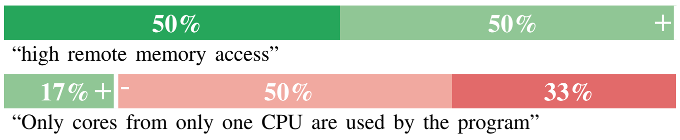

Figure 3: Levels of agreement on observations related to the "**Sharing of data between CPUs on NUMA systems**" problem.

Figures 3 to 5 and 7 to 16, together with Appendix 1, 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. For example, in Figure 7 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.

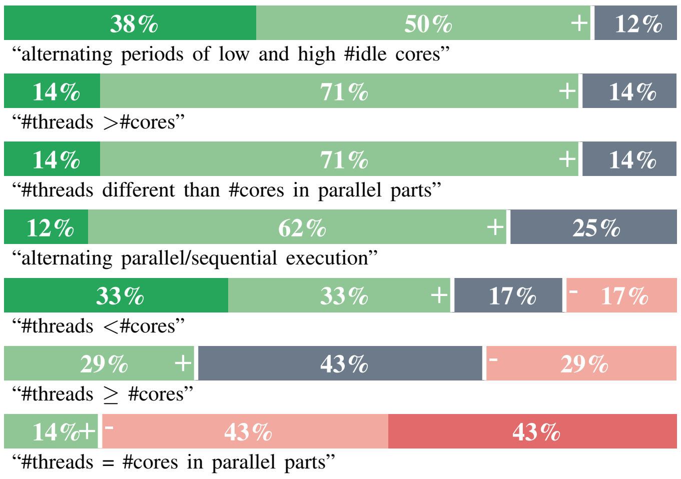

Figure 4: Levels of agreement on observations related to the "**Bad threads to cores ratio**" problem.

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.

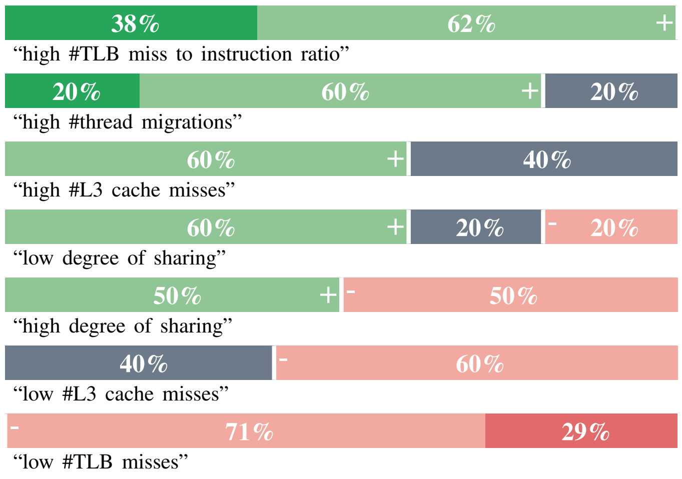

Figure 5: Levels of agreement on observations related to the "**TLB Locality**" problem.

In the next section we discuss the results of this study, together with the results of the survey presented in section 4.

6 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 with respect to the model and taxonomy.

Figure 6 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 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 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

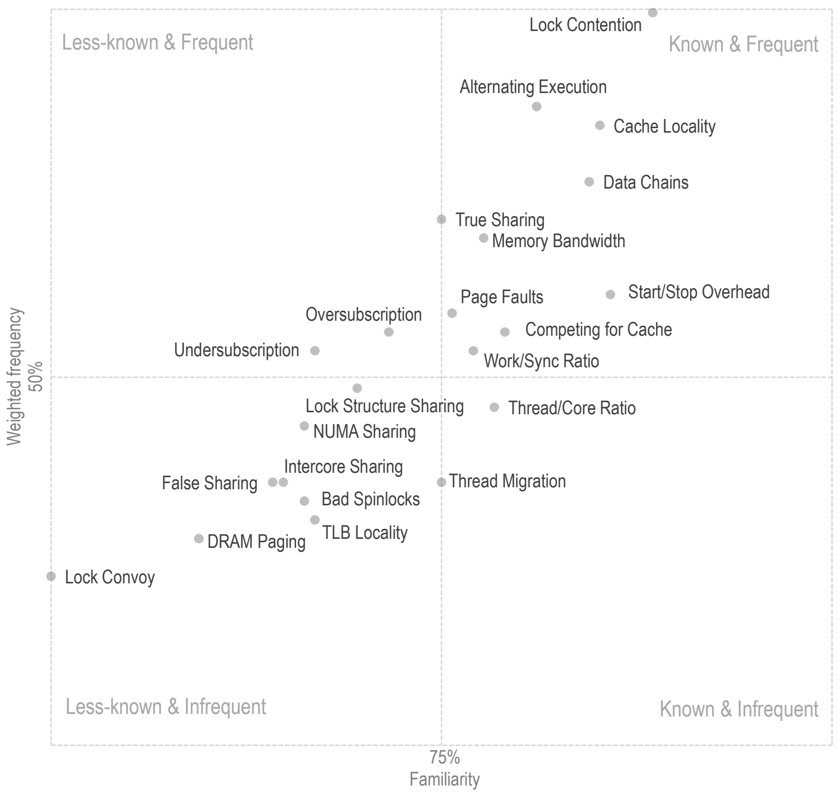

Figure 6: Quadrant plot of parallel performance problems mapped against Familiarity and Frequency.

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.

6.1 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 7). 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.: [78], [95]). 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 [78].

Poor cache locality (Figure 16). 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 [70]. This problem can be diagnosed by using low-level hardware perfor-

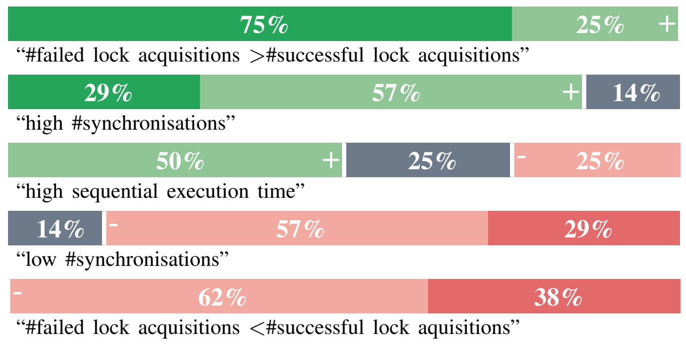

Figure 7: Levels of experts' agreement on observations related to the "**Lock contention**" problem.

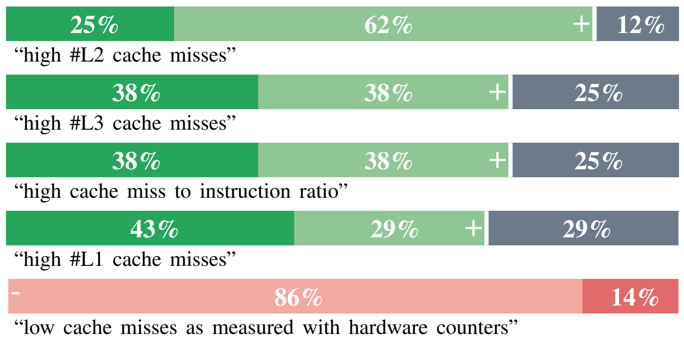

Figure 8: Levels of experts' agreement on observations related to the "**Cache Locality**" problem.

mance counters[7] which can be accessed via tools such as Intel VTune or Intel Performance Counter Monitor, for Intel-family 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 9). 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 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.

7. 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.

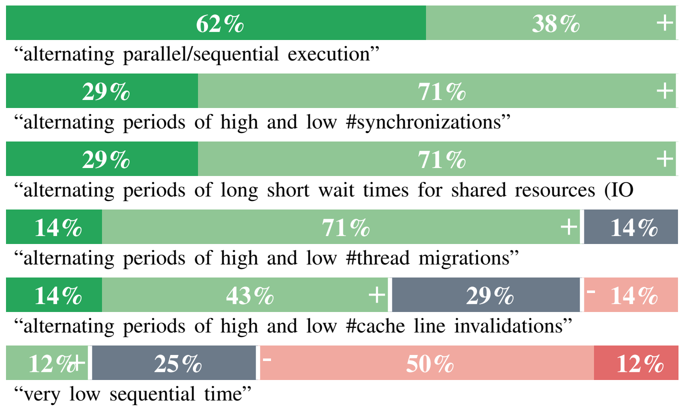

Figure 9: Levels of experts' agreement on observations related to the "**Alternating sequential/parallel execution**" problem.

Chains of data dependencies with too little parallelism (Figure 10). 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 [23], 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 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 [59]. 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 [59], [96]. This would require a higher-level approach to understanding performance problems, based on the parallel orchestration model.

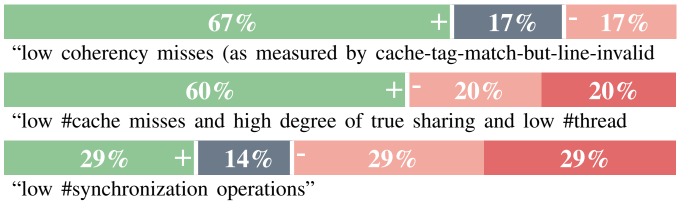

Figure 10: Levels of agreement on observations related to the "**Chains of data dependencies, too little parallelism**" problem.

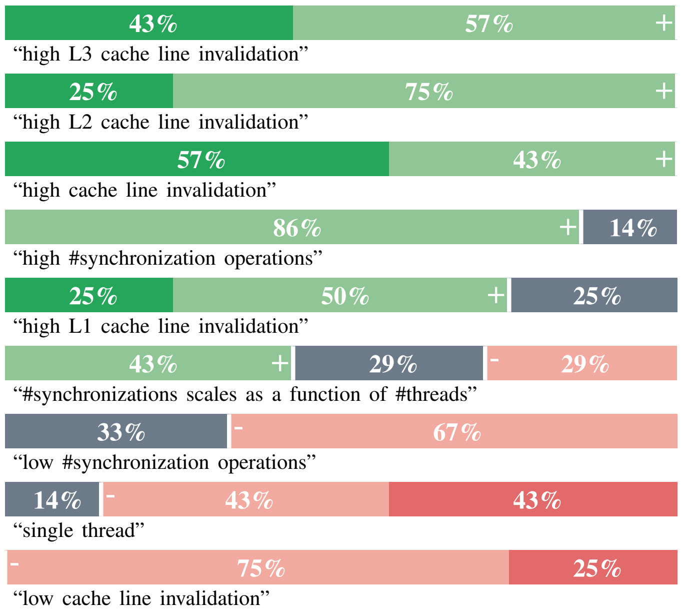

Figure 11: Levels of experts' agreement on observations related to the "**True sharing of updated data**" problem.

6.2 Less-Known but Frequent

As we mentioned before, the parallel performance problems in figure 6 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 11). 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, 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.

Undersubscription The undersubscription problem (Figure 12) 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.

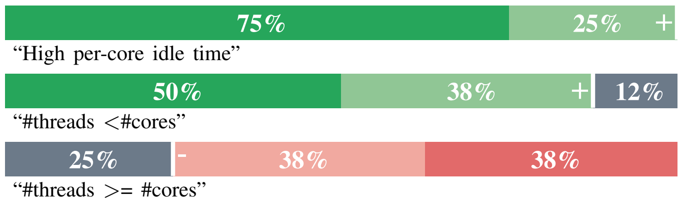

Figure 12: Levels of agreement on observations related to the "**Undersubscription**" problem.

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 in Figure 13, 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.

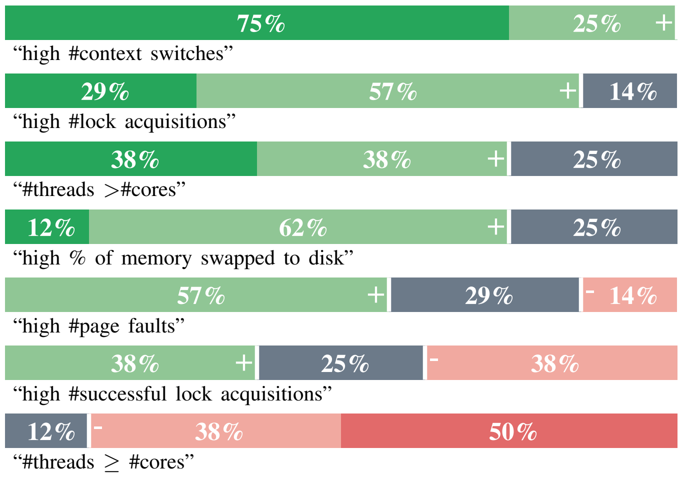

Figure 13: Levels of agreement on observations related to the "**Oversubscription**" problem.

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 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.

6.3 Less-known and Infrequent

The less-known and infrequent parallel performance problems of figure 6 tend to be rather technical and obscure. As mentioned at the start of section 6 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.

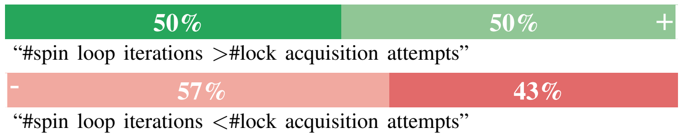

Figure 14: Levels of agreement on observations related to the "**Badly-behaved spinlocks**" problem.

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 14). There is also a much agreement (although not unanimity) about the observations relating to false data sharing (fig. 15). This is in agreement with the literature, where cache invalidation is clearly identified as a key symptom for true/false sharing [97]. Similarly, the experts broadly agree on the observations that might indicate a problem with translation look-aside buffer (TLB) locality (fig. 5), which is another problem that can have a large impact on execution time.

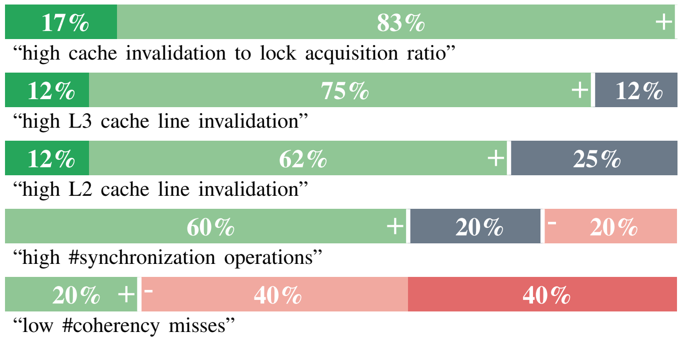

Figure 15: Levels of agreement on observations related to the "**False data sharing**" problem.

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.

6.4 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 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.

6.5 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 associated 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 [32], [48]. 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 6.2 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 developer, 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 16), 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.

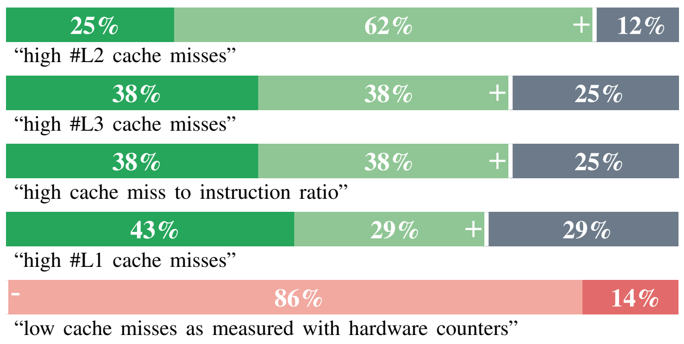

Figure 16: Levels of agreement on observations related to the “**Cache Locality**” problem.

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.

7 Conclusions

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 multicore 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. Our experience of teaching parallel programming over several years is 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.

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.

Acknowledgements

This work was supported by supported, in part, by Science Foundation Ireland grant 10/CE/I1855.

References

[1] K. D. Bosschere, "Upcoming computing system challenges - the HiPEAC vision," *it - Information Technology*, vol. 50, no. 5, pp. 285–292, 2008. [Online]. Available: http://dx.doi.org/10.1524/itit.2008.0497
[2] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.
[3] M. Levy and T. M. Conte, "Embedded multicore processors and systems," *IEEE Micro*, vol. 29, no. 3, pp. 7–9, May 2009. [Online]. Available: http://dx.doi.org/10.1109/MM.2009.41
[4] S. Akhter and J. Roberts, *Multi-core programming: Increasing Performance through Software Multi-threading*. Intel press Hillsboro, 2006, vol. 33.
[5] C. Breshears, *The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications*. O'Reilly Media, Inc., 2009.
[6] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, ser. AFIPS '67 (Spring). New York, NY, USA: ACM, 1967, pp. 483–485. [Online]. Available: http://doi.acm.org/10.1145/1465482.1465560
[7] H. W. Meuer, E. Strohmaier, J. Dongarra, and H. D. Simon, *The TOP500: History, Trends, and Future Directions in High Performance Computing*, 1st ed. Chapman & Hall/CRC, 2014.
[8] M. Duranton, S. Yehia, B. De Sutter, K. De Bosschere, A. Cohen, B. Falsafi, G. Gaydadjiev, M. Katevenis, J. Maebe, H. Munk, N. Navarro, A. Ramirez, O. Temam, and M. Valero, "The HiPEAC Vision, 2010," pp. 1–60, 2010.
[9] P. McKenney, M. Gupta, M. Michael, P. Howard, J. Triplett, and J. Walpole, "Is parallel programming hard, and if so, why?" *Control*, 2002.
[10] A. J. Viera, J. M. Garrett *et al.*, "Understanding interobserver agreement: the kappa statistic," *Fam Med*, vol. 37, no. 5, pp. 360–363, 2005.
[11] K. Asanovic, J. Wawrzynek, D. Wessel, K. Yelick, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, and K. Sen, "A view of the parallel computing landscape," *Communications of the ACM*, vol. 52, no. 10, p. 56, Oct. 2009.
[12] R. Bryant and D. O'Hallaron, *Computer Systems: A Programmer's Perspective*. Addison-Wesley, 2010.
[13] A. Nistor, L. Song, D. Marinov, and S. Lu, "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.
[14] S. Han, Y. Dang, S. Ge, and D. Zhang, "Performance Debugging in the Large via Mining Millions of Stack Traces," in *ICSE '12*, 2012, pp. 176–186.
[15] A. Ko and B. Myers, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Transactions on Software Engineering*, vol. 32, no. 12, pp. 971–987, 2006.
[16] T. Fritz and G. C. Murphy, "Using information fragments to answer the questions developers ask," *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, vol. 1, p. 175, 2010.
[17] A. Begel and T. Zimmermann, "Analyze This! 145 Questions for Data Scientists in Software Engineering," in *ICSE '14*, 2014, pp. 12–23.
[18] K.-j. Stol and B. Fitzgerald, "Two's Company, Three's a Crowd: A Case Study of Crowdsourcing Software Development," *ICSE '14*, pp. 187–198, 2014.
[19] N. Carolina, E. Murphy-hill, T. Zimmermann, and N. Nagappan, "Cowboys, ankle sprains, and keepers of quality: How is video game development different from software development?" in *ICSE '14*, 2014, pp. 1–11.
[20] P. Pirolli and S. Card, "Information foraging in information access environments," *Proceedings of the SIGCHI conference on Human factors in computing systems - CHI '95*, pp. 51–58, 1995. [Online]. Available: http://portal.acm.org/citation.cfm?doid=223904.223911
[21] J. Lawrance, C. Bogart, M. Burnett, S. Member, R. Bellamy, K. Rector, and S. D. Fleming, "How Programmers Debug, Revisited: An Information Foraging Theory Perspective," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 197–215, 2013.
[22] D. Piorkowski and S. Fleming, "The whats and hows of programmers' foraging diets," *CHI '13 Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 3063–3072, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2466418
[23] R. Atachiants, D. Gregg, K. Jarvis, and G. Doherty, "Design considerations for parallel performance tools," in *CHI '14 Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2014, pp. 2501–2510. [Online]. Available: http://dl.acm.org/citation.cfm?id=2557350
[24] B. Beizer, *Software Testing Techniques (2Nd Ed.)*. Van Nostrand Reinhold Co., 1990.
[25] M. I. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation Table of Contents*. MIT Press Cambridge, MA, USA, 1991.

[26] S. L. P. J. Trinder, P. W., K. Hammond, H.-W. Loidl, "Algorithm + strategy = parallelism," *Journal of Functional Programming*, vol. 8, no. 1, pp. 23–60, 1998.

[27] B. R. P. Dhrubajyoti Goswami, Ajit Singh, "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.

[28] S. M. A. Laksberg, H. Sutter, A. Robison, "A C ++ Library Solution to Parallelism," INCITS, InterNational Committee for Information Technology Standards, Tech. Rep., 2012.

[29] D. K. G. Campbell, "Towards the Classification of Algorithmic Skeletons," Tech. Rep., 1996.

[30] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*, 1st ed. OReilly Media, Inc, 2007.

[31] K. K. Ryan Newton, Frank Schlimbach, Mark Hampton, "Capturing and Composing Parallel Patterns with Intel CnC," in *HotPar '10*, 2010.

[32] T. Fahringer, *Automatic Performance Prediction of Parallel Programs*, 1st ed. Springer Publishing Company, Incorporated, 2011.

[33] N. Siegmund, S. S. Kolesnikov, K. Christian, S. Apel, and G. Saake, "Predicting Performance via Automated Feature-Interaction Detection," in *ICSE '12*, no. ii, 2012, pp. 167–177.

[34] E. Ipek, B. de Supinski, and M. Schulz, "An approach to performance prediction for parallel applications," *Euro-Par 2005 Parallel*, pp. 196–205, 2005. [Online]. Available: http://www.springerlink.com/index/ay63wtdah19m036g.pdf

[35] L. Yang and F. Mueller, "Cross-Platform Performance Prediction of Parallel Applications Using Partial Execution," *ACM/IEEE SC 2005 Conference (SC'05)*, pp. 40–40, 2005. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1559992

[36] J. Liu, D. Nicol, and B. Premore, "Performance Prediction of a Parallel Simulator," *Distributed Simulation,*, 1999. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=766172

[37] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. W. Jr, "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.

[38] M. Jovic, A. Adamoli, and M. Hauswirth, "Catch Me If You Can : Performance Bug Detection in the Wild," in *OOPSLA*, 2011, pp. 155–170.

[39] S. Zaman, B. Adams, and A. E. Hassan, "A Qualitative Study on Performance Bugs," in *MSR*, 2012, pp. 199–208.

[40] C. Sadowski and A. Shewmaker, "The Last Mile: Parallel Programming and Usability," *FOSER*, 2010.

[41] T. Mattson and M. Wrinn, "Parallel programming: can we PLEASE get it right this time?" *Proceedings of the 45th annual Design Automation ...*, pp. 7–11, 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1391474

[42] L. Snyder, "Parallel programming and the poker programming environment." DTIC Document, Tech. Rep., 1984.

[43] L. Snyder and D. Socha, "Poker on the cosmic cube: The first retargetable parallel programming language and environment." DTIC Document, Tech. Rep., 1986.

[44] M. Heath, "Visualizing the performance of parallel programs," *Software, IEEE*, pp. 29–39, 1991. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=84214

[45] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall, "The Paradyn parallel performance measurement tool," *Computer*, vol. 28, no. 11, pp. 37–46, 1995. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=471178

[46] T. Casavant, "Tools and Methods for Visualization of Parallel Systems and Computations Guest Editors Introduction," pp. 103–104, Jun. 1993. [Online]. Available: http://linkinghub.elsevier.com/retrieve/doi/10.1006/jpdc.1993.1049

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

[48] K. A. Huck and A. D. Malony, "PerfExplorer: A performance data mining framework for large-scale parallel computing," in *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, ser. SC '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 41–. [Online]. Available: http://dx.doi.org/10.1109/SC.2005.55

[49] J. Reinders, *VTune performance analyzer essentials*. Intel Press, 2005.

[50] P. J. Drongowski, A. C. Team, and B. D. Center, "An introduction to analysis and optimization with amd codeanalyst performance analyzer," *Advanced Micro Devices, Inc*, 2008.

[51] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, and J. Yang, "Visualizing the execution of java programs," in *Software Visualization*. Springer, 2002, pp. 151–162.

[52] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Evaluating the accuracy of Java profilers," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '10. New York, NY, USA: ACM, 2010, pp. 187–197. [Online]. Available: http://doi.acm.org/10.1145/1806596.1806618

[53] I. Park and M. K. Raghuraman, "Server diagnosis using request tracking," in *1st Workshop on the Design of Self-Managing Systems, held in conjunction with DSN 2003*, 2003.

[54] R. Jain, *The art of computer systems performance analysis - techniques for experimental design, measurement, simulation, and modeling.* Wiley, 1991.

[55] K. C. Foo, Z. M. J. Jiang, B. Adams, A. E. Hassan, Y. Zou, and P. Flora, "An industrial case study on the automated detection of performance regressions in heterogeneous environments," in *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 159–168. [Online]. Available: http://dl.acm.org/citation.cfm?id=2819009.2819034

[56] T.-H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Detecting performance anti-patterns for applications developed using object-relational mapping," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 1001–1012. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568259

[57] M. Litoiu and C. Barna, "A performance evaluation framework for web applications," *Journal of Software: Evolution and Process*, vol. 25, no. 8, pp. 871–890, 2013. [Online]. Available: http://dx.doi.org/10.1002/smr.1563

[58] "Intel 64 and IA-32 architectures optimization reference manual, order no. 248966-030," Intel Corporation, Tech. Rep., September 2014.

[59] "Intel threading building blocks design patterns, document number 323512-003us," Intel Corporation, Tech. Rep., September 2010.

[60] "The MSDN common patterns for poorly-behaved multithreaded applications." [Online]. Available: https://msdn.microsoft.com/en-us/library/ee329530.aspx

[61] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, and N. R. Tallent, "HPCToolkit: Tools for performance analysis of optimized parallel programs," *Context*, pp. 1–7, 2010.

[62] M. Burtscher, B.-D. Kim, J. Diamond, J. McCalpin, L. Koesterke, and J. Browne, "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*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.41

[63] M. Süß and C. Leopold, "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.

[64] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV. New York, NY, USA: ACM, 2010, pp. 129–142. [Online]. Available: http://doi.acm.org/10.1145/1736020.1736036

[65] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel Programming in OpenMP*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.

[66] M. Ben-Ari, *Principles of Concurrent Programming*. Prentice Hall Professional Technical Reference, 1982.

[67] K. Russell and D. Detlefs, "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*, ser. OOPSLA '06. New York, NY, USA: ACM, 2006, pp. 263–272. [Online]. Available: http://doi.acm.org/10.1145/1167473.1167496

[68] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[69] K. Goto and R. A. v. d. Geijn, "Anatomy of high-performance matrix multiplication," *ACM Trans. Math. Softw.*, vol. 34, no. 3, pp. 12:1–12:25, May 2008. [Online]. Available: http://doi.acm.org/10.1145/1356052.1356053

[70] U. Drepper, "What Every Programmer Should Know About Memory," Red Hat, Tech. Rep., 2007.

[71] R. H. B. Netzer and B. P. Miller, “What are race conditions?: Some issues and formalizations,” *ACM Lett. Program. Lang. Syst.*, vol. 1, no. 1, pp. 74–88, Mar. 1992. [Online]. Available: http://doi.acm.org/10.1145/130616.130623

[72] M. Bishop, M. Dilger *et al.*, “Checking for race conditions in file accesses,” *Computing systems*, vol. 2, no. 2, pp. 131–152, 1996.

[73] D. Engler and K. Ashcraft, “RacerX: Effective, static detection of race conditions and deadlocks,” in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP ’03. New York, NY, USA: ACM, 2003, pp. 237–252. [Online]. Available: http://doi.acm.org/10.1145/945445.945468

[74] J. Mak and A. Mycroft, “Limits of parallelism using dynamic dependency graphs,” in *Proceedings of the Seventh International Workshop on Dynamic Analysis*, ser. WODA ’09. New York, NY, USA: ACM, 2009, pp. 42–48. [Online]. Available: http://doi.acm.org/10.1145/2134243.2134253

[75] C. Iancu, S. Hofmeyr, F. Blagojevic, and Y. Zheng, “Oversubscription on multicore processors,” in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, April 2010, pp. 1–11.

[76] T. Constantinou, Y. Sazeides, P. Michaud, D. Fetis, and A. Seznec, “Performance implications of single thread migration on a chip multi-core,” *SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 80–91, Nov. 2005.

[77] R. Odaira, J. G. Castanos, and H. Tomari, “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*, ser. PPoPP ’14. New York, NY, USA: ACM, 2014, pp. 131–142. [Online]. Available: http://doi.acm.org/10.1145/2555243.2555247

[78] N. Tallent, J. Mellor-Crummey, and A. Porterfield, “Analyzing lock contention in multithreaded applications,” in *PPoPP ’10: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1693489

[79] M. Blasgen, J. Gray, M. Mitoma, and T. Price, “The convoy phenomenon,” *SIGOPS Oper. Syst. Rev.*, vol. 13, no. 2, pp. 20–25, Apr. 1979. [Online]. Available: http://doi.acm.org/10.1145/850657.850659

[80] T. Anderson, “The performance of spin lock alternatives for shared-money multiprocessors,” *Parallel and Distributed Systems, IEEE Transactions on*, vol. 1, no. 1, pp. 6–16, Jan 1990.

[81] M. S. Papamarcos and J. H. Patel, “A low-overhead coherence solution for multiprocessors with private cache memories,” in *Proceedings of the 11th Annual International Symposium on Computer Architecture*, ser. ISCA ’84. New York, NY, USA: ACM, 1984, pp. 348–354. [Online]. Available: http://doi.acm.org/10.1145/800015.808204

[82] W. J. Bolosky, M. L. Scott, R. P. Fitzgerald, R. J. Fowler, and A. L. Cox, “NUMA policies and their relation to memory architecture,” *SIGARCH Comput. Archit. News*, vol. 19, no. 2, pp. 212–221, Apr. 1991. [Online]. Available: http://doi.acm.org/10.1145/106975.106994

[83] R. Rajwar and J. R. Goodman, “Speculative lock elision: Enabling highly concurrent multithreaded execution,” in *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 34. Washington, DC, USA: IEEE Computer Society, 2001, pp. 294–305. [Online]. Available: http://dl.acm.org/citation.cfm?id=563998.564036

[84] Y. Zhang, M. Kandemir, and T. Yemliha, “Studying inter-core data reuse in multicores,” in *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS ’11. New York, NY, USA: ACM, 2011, pp. 25–36. [Online]. Available: http://doi.acm.org/10.1145/1993744.1993748

[85] W. Heirman, T. Carlson, K. Van Craeynest, I. Hur, A. Jaleel, and L. Eeckhout, “Undersubscribed threading on clustered cache architectures,” in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, Feb 2014, pp. 678–689.

[86] M. Wolfe and U. Banerjee, “Data dependence and its application to parallel processing,” *Int. J. Parallel Program.*, vol. 16, no. 2, pp. 137–178, Apr. 1987. [Online]. Available: http://dx.doi.org/10.1007/BF01379099

[87] J. Lee, H. Wu, M. Ravichandran, and N. Clark, “Thread tailor: Dynamically weaving threads together for efficient, adaptive parallel applications,” in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA ’10. New York, NY, USA: ACM, 2010, pp. 270–279. [Online]. Available: http://doi.acm.org/10.1145/1815961.1815996

[88] K. Kennedy and K. S. McKinley, “Optimizing for parallelism and data locality,” in *ACM International Conference on Supercomputing 25th Anniversary Volume*. New York, NY, USA: ACM, 2014, pp. 151–162. [Online]. Available: http://doi.acm.org/10.1145/2591635.2667164

[89] G. B. Kandiraju and A. Sivasubramaniam, “Characterizing the d-TLB behavior of SPEC CPU2000 benchmarks,” *SIGMETRICS Perform. Eval. Rev.*, vol. 30, no. 1, pp. 129–139, Jun. 2002. [Online]. Available: http://doi.acm.org/10.1145/511399.511351

[90] J. Fotheringham, “Dynamic storage allocation in the atlas computer, including an automatic use of a backing store,” *Commun. ACM*, vol. 4, no. 10, pp. 435–436, Oct. 1961. [Online]. Available: http://doi.acm.org/10.1145/366786.366800

[91] N. R. Mahapatra and B. Venkatrao, “The processor-memory bottleneck: Problems and solutions,” *Crossroads*, vol. 5, no. 3es, Apr. 1999. [Online]. Available: http://doi.acm.org/10.1145/357783.331677

[92] S. Kim, D. Chandra, and Y. Solihin, “Fair cache sharing and partitioning in a chip multiprocessor architecture,” in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT ’04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 111–122. [Online]. Available: http://dx.doi.org/10.1109/PACT.2004.15

[93] J. Torrellas, M. Lam, and J. L. Hennessy, “False sharing and spatial locality in multiprocessor caches,” *Computers, IEEE Transactions on*, vol. 43, no. 6, pp. 651–663, Jun 1994.

[94] J. R. Landis and G. G. Koch, “The measurement of observer agreement for categorical data,” *Biometrics*, vol. 33, pp. 159–174, 1977.

[95] Y. Huang, Z. Cui, L. Chen, and W. Zhang, “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. [Online]. Available: http://dl.acm.org/citation.cfm?id=2370854

[96] S. Siu, M. D. Simone, D. Goswami, and A. Singh, “Design Patterns for Parallel Programming,” *PDPTA*, 1996. [Online]. Available: http://www.cs.uiuc.edu/homes/snir/PPP/skeleton/dpndp.pdf

[97] G. Venkataramani, C. J. Hughes, S. Kumar, and M. Prvulovic, “Deft: Design space exploration for on-the-fly detection of coherence misses,” *ACM Trans. Archit. Code Optim.*, vol. 8, no. 2, pp. 8:1–8:27, Jun. 2011. [Online]. Available: http://doi.acm.org/10.1145/1970386.1970389