Design Considerations for Parallel Performance Tools

Roman Atachiants, David Gregg, Kim Jarvis and Gavin Doherty
Lero@TCD
Trinity College Dublin
atachiar@tcd.ie, David.Gregg@tcd.ie, kjarvis@tcd.ie, Gavin.Doherty@tcd.ie

ABSTRACT

In recent years there has been a shift in microprocessor manufacture from building single-core processors towards providing multiple cores on the same chip. This shift has meant that a much wider population of developers are faced with the task of developing parallel software: a difficult, time consuming and expensive process. With the aim of identifying issues, emerging practices and design opportunities for support, we present in this paper a qualitative study in which we interviewed a range of software developers, in both industry and academia. We then perform a systematic analysis of the data and identify several cross-cutting themes. These analysis themes include the practical relevance of the probe effect, the significance of orchestration models in development and the mismatch between currently available tools and developers' needs. We also identify an important characteristic of parallel programming, where the process of optimisation goes hand in hand with the process of debugging, as opposed to clearer distinctions which may be made in traditional programming. We conclude with reflection on how the study can inform the design of software tools to support developers in the endeavour of parallel programming.

Author Keywords

many-core, multi-core; parallel programming; qualitative study;visualisation

ACM Classification Keywords

H.5.m. Information Interfaces and Presentation (e.g. HCI): Miscellaneous

INTRODUCTION

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

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

In the near future, computers are expected to have even more cores - the trend towards "many-core" computing. A many-core processor is a multi-core processor in which the number of cores is large enough that traditional multiprocessor techniques are no longer efficient. While with a small number of cores performance gains can be achieved simply by running different programs simultaneously, with many cores, performance gains will only be achieved through the use of parallel programming. On modern servers and supercomputers it is already common to have hundreds or even thousands of cores.

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

A programmer seeking to parallelise a program has to overcome the challenges of synchronisation, non-determinism and orchestration that a programmer writing an equivalent sequential program would not have to face. Additionally, the very process of parallelising may introduce bugs, deadlocks and race conditions into the program. On top of that, when looking at a program it is not necessarily obvious what its parallel performance will be.

For over two decades, a great deal of research effort has been directed at tools for improving the performance of parallel applications and over 200 now defunct, parallel-programming languages saw the light in the 1990s [29]. However, twenty years later, concurrency bugs are still extremely common and theoretical performance is often very difficult to realise [38].

CHI 2014, April 26–May 1, 2014, Toronto, Ontario, Canada.

ACM 978-1-4503-2473-1/14/04..$15.00.
http://dx.doi.org/10.1145/2556288.2557350

In this paper a qualitative study of parallel programming practices is presented. The goal of the study is to increase our understanding of the human factors and practices involved in parallel programming tasks. By doing so, we hope to shed light on some of the challenges which have been previously noted in the literature [19, 38]. We hope to gain a better understanding of the underlying techniques and processes the programmers of concurrent systems use in order to stay productive and inform the design of tools and visualisation systems that could help programmers in their endeavours.

Following a review of related work, we describe the methodology used for the study and present a qualitative analysis of the data. We propose a number of possible implications for the design of support tools, and look towards the development of a set of design guidelines based on the lessons gleaned from the interviews and analysis.

RELATED WORK

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

That being said, over the past twenty years many studies have been carried out on novice programmers and identified good and bad aspects of today's programming systems [33]. While testing and debugging are two very complex areas for novice programmers, some researchers claim that programming tools should support source-level debugging with data visualisation to be more effective [4].

Using an appropriate concrete metaphor, a familiar analogy explaining how the programming system works, can have a positive effect on the usability [27] and also maximises transfer of knowledge if the metaphor is close to a real-world system [41].

Expert programming can be considered as an opportunistic activity [13], however some research indicates that expert programmers use intricate plans and strategies in order to schedule and prioritise their activities [1]. Planning is common among novice programmers and the absence of good planning strategy results in wrong assumptions and more bugs [3]. Programming tools should also consider locality; related components should be physically close [2] and hidden dependencies reduce understanding [13].

For both novice and expert programmers the ability to test partial solutions is an important feature [33]. Incremental running and testing, a programming strategy where the code is iteratively tested while new code is being written, have been found to be an effective debugging strategy [14], and it has been observed that developers perform better when such a strategy is adopted [12, 13].

In addition, there have been attempts to understand how programmers work from a sensemaking perspective, applying information foraging [35] to understand how developers debug [23, 34]. Various empirical studies and models have attempted to understand and help answer questions that developers' ask [20, 10]. These have used a variety of methodologies, including observations of pairs of programmers given sample tasks [39], and questionnaire based studies [22].

Prabhu et al. [36] present an extensive survey of software practices in computational science. They conclude that current programming systems and tools do not meet the requirements for scientific computing. They indicate that most tools assume that programmers would invest time and energy to learn and master a particular system, which turned out not to be the case with programmers wanting results immediately. They found that most of the scientists understand the importance of parallel programming in the context of scientific computing and many scientists spend a significant amount of time and energy programming. Despite this effort, most scientists seem to be unsatisfied with the performance of their programs and believe that the improvement of performance will significantly improve their research and, in some cases, allow larger experiments or enable fundamentally new research avenues. The survey concluded that overall the needs of computational scientists were underserved and new tools and techniques were needed to unlock the potential of high-performance computing.

In another study, Hannay et al. conducted an online survey of approximately two thousand scientists in order to study software engineering practices in the scientific community [16]. Their study found that for the concepts "software testing" and "software verification" scientists assign on average a higher level of importance to these concepts than they judge their level of understanding of these concepts. Scientists seem to care a great deal about the correctness of their code, but feel they lack knowledge of software engineering practices to verify it. Additionally, their findings suggest that informal learning from peers was more important to scientists than formal education at an academic institution or formal training at the work place, they postulate that this may be due to the lack of appropriate formal training. Their study also confirmed the hypothesis that the majority of scientists use desktop computers most of the time for developing and using scientific software as opposed to supercomputers or clusters.

A classroom study conducted across a number of universities compared different programming models across a variety of representative applications [18]. It is worth noting that most of these studies use novices as the study participants rather than experienced programmers. The study compared shared memory (multi-core) and message passing (distributed) implementations of two problems, written by novice programmers. They found that the message passing development effort, measured in lines of code, was statistically greater but resulted into more correct programs. On the other hand, shared memory programs were smaller and easier to write but were more error-prone.

Luff conducted an experiment comparing developers' performance using various parallel programming models (actor model, transactional memory and standard shared memory threading with locks) while keeping the programming language and environment (IDE) the same [25]. The results were

inconclusive and showed no significant difference in any objective measurement between those models.

A study conducted by Eccles et al. [8] had novice and expert programmers both categorising different parallel algorithms, using a card sorting method. They found that novices and experts used a different classification scheme: novice programmers organised problems around the problem domain while expert programmers organised problems around communication granularity and overhead. This difference in classification could identify a set of concepts which delineate novices from experts [38]. They concluded that the best way of organising parallel programming material and libraries is around the expert classification scheme, and postulate that it would result in a more usable parallel software.

A wide variety of prior literature related to program maintenance focuses on debugging, with a particular emphasis on novice versus expert difference [28]. Most research agrees that reading and understanding code is the fastest debugging method [7, 43], although this is not always feasible for very large programs [21]. Fix et al. [9] found that experts had more sophisticated mental models than novices, and so were able to use them more effectively to debug programs.

A study by Pancake [32] attempted to determine a correlation between mental models and the effectiveness of visualisations for parallel debuggers. It demonstrated that it is possible to implement various conceptual models using any programming language, however program development becomes significantly easier and more reliable when the language has support for expressing the desired model. The same correlation applies to the debugger visualisation models.

Fleming [37] conducted an exploratory think-aloud study in which he observed 15 programmers debugging multi-threaded server application which was seeded with a defect. He claimed that the programmers who succeeded used a previously undocumented failure-trace strategy while debugging, and using such a strategy made the programmers more likely to succeed. The strategy involved modelling interactions between various threads in the program in order to find a failure trace (i.e. the interaction that led to a failure). He also postulated that cognitive strain may have been an important barrier as the failure-trace strategy was modelled internally. In addition, it is claimed that the inherent concurrency of parallel programming makes managing hypotheses regarding the cause of a bug more difficult.

An important aspect of tool support for multi-core programming is understanding performance data. Given the volume and complexity of this data, visualisation is an important design direction as it leverages capacities and bandwidth of the visual system to quickly assess and understand large volumes of data. Visualising the behaviour of parallel programs is a very complex task, as the behaviour of the programs themselves is often complex. The area of effective visualisation techniques for parallel programs is still relatively unexplored within parallel programming research and has usability implications [38]. However, the need to form a scientific body of research, develop human-centered models, and target production level applications and their developers has been recognised [26]. A number of tools such as ParaGraph and ParaDyn have been developed to visualise behaviour [17, 30]; most of the tools that have been developed for the High-Performance Computing domain 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 problems faced in the construction of parallel programs. With regards to tools, existing systems can be seen as providing answers to two main issues: topological mapping, and synchronisation. The topology issue requires that spatial relationships in programs be understood. The synchronisation issue requires various events occurring within the processor, to be correlated [6]. While some existing visualisations are potentially useful, there is a need for analysis of how such tools can aid in the diagnosis of problems.

METHODOLOGY

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

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

Overall, 22 people were interviewed. 14 of them were recorded, resulting in over 8 hours of audio. Interviews were transcribed (around 47,000 words), open coded (582 open codes, unified into 252 codes), categorised in 8 major categories and sub-categorised in 23 sub-categories. Interview participants are summarised in Table 1.

Interviews were semi-structured, and interviewees were asked to talk about the challenges they had to overcome in their every day work related to parallel programming, the tools they had employed, and practices they used. While this analysis method has its roots in Grounded Theory [42], we would like to highlight that it was not our goal to build a holistic theory from the data. Following common practice in HCI [31], we used the approach as the foundation

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

Table 1. A table of participants with their years of experience, main activity and the type of organisation.

for a focussed analysis of the transcribed interviews. Interview questions along with the full set of codes are available at `https://www.scss.tcd.ie/ManyCore/`.

DEVELOPING PARALLEL SOFTWARE

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

Context for development

The developers interviewed were mostly based in industry and every organisation had their own unique set of constraints and concerns. Their working environments (both physical and virtual) are composed of various components such as software specifications, budgeting and deadlines, human resources, hardware and software constraints, programming languages, software architecture, the actual code, etc. An interesting aspect of the study is that the programmers interviewed didnt dwell on programming paradigms and talked about their experience across different programming languages. We know, however, that many were using imperative languages such C, C++, fortran or Java, and some used hybrid languages such as C# or Matlab.

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

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

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

While developers often worked with extensive tool support in the form of IDE's, this support does not extend specifically to debugging and optimising parallel software. As a result, developers tend to rely on intuition and resort to 'thinking really hard' about the problem. There are two aspects to reasoning about problems; on a conceptual level developers want to focus on the problematic elements of their program and abstract away elements not immediately relevant to the matter in hand. At a concrete level, there are also issues of experimental noise; they may wish to replicate the problem in the simplest possible form in order to clearly identify the source of the issue and test possible solutions.

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

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

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

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

Understanding

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

In order to improve their understanding, developers use a range of techniques. Some of these techniques are only pos-

sible with very specific tools whereas in other cases they may simply ask other people.

Table 2 provides a non-exhaustive list of techniques that were mentioned or discussed to some extent during the interviews.

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

Table 2. A non-exhaustive set of techniques for understanding and the percentage of interviewees who were talking about the subject.

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

A developer in a large organisation commented:

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

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

Inadequate Tools

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

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

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

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

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

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

The Probe Effect

One of the most problematic barriers for understanding the performance of complex software systems, is the presence of the probe effect. The probe effect denotes an unintended change to the behaviour of a system caused by observing (measuring) that system [11, 24]. In order to measure something in the program, an automated tool or a person adds additional code to record features of interest. This approach is called program instrumentation.

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

In order for a performance measurement system to be safe to use on production systems, it is generally considered that the performance analysis infrastructure should have zero probe effect when disabled and should not accidentally induce any errors when enabled [5]. However, most instrumentation techniques have a probe effect when enabled making it difficult to apply continuous monitoring of performance on production systems.

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

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

Information representation and resources

In order to cope with the complexity of software development, developers create abstract representations of the environment and contextualise the work they are performing. For example, interviewees talk about a global scope or a local

scope. The scope represents a context in which they perform their tasks, allowing them to focus and reason on a particular level:

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

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

As mentioned above, complexity can be overwhelming. Developers therefore use tools to offload knowledge into the world. For example, several developers reported using UML[1] or sketching tools, even pen and paper, in order to externalise their knowledge by 'charting' the environment in this way:

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

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

> Interviewee: *"I'll often look to use the Intel MKL[2] libraries, wherever possible. [...] But the documentation for it is really excellent. "*

Developers seek information relevant for the task at hand. For example, if a programmer designs a new feature which complements a bigger system, they need information about the existing elements of the system in order to achieve their goal. If the programmer develops a particular algorithm destined to work on particular hardware, they might need hardware specifications in order to create an optimal algorithm. The programmer needs information relevant to their task, but information is often fragmented and distributed across various sources (documentation, internal wiki, people).

Orchestration

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

Orchestration Models

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

> Interviewee: *"The goal was to decouple an image processing part as much as possible in a pipeline and partition the tasks on the different resources [machines]."*

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

Across these various projects and solutions that were put in place, similar orchestration patterns were applied. Surprisingly only some of the developers were familiar with formalised design patterns and named them. However, many developers we interviewed were essentially talking about a design pattern. Table 3 shows the percentage of developers we interviewed who were using knowingly or unknowingly a recognisable design pattern in their work.

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

Table 3. A non-exhaustive set of orchestration models and the percentage of interviewees who were talking about the subject.

For example, one of most commonly mentioned design patterns is the Producer/Consumer, a design pattern that is used

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

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

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

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

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

Orchestration models are a type of design pattern employed in parallel programming, and they span different contexts (or scopes) on which developers are focusing. In software engineering design patterns are often related to reusable code skeletons for quick and reliable development of parallel applications [40]. In contrast, orchestration models are more abstract, spanning not only patterns related to software architecture, but hardware and system architectures as well.

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

Difficulties with Orchestration

While the orchestration models provide an initial basis for design, helping the programmers to reason about and implement their system, there are many issues associated with the process of orchestration. In this section we highlight some of the difficulties described by the developers.

To illustrate our point, we start by giving a couple of examples of problems the programmers encountered.

Problem: Novice programmers (computer science students) were implementing a concurrent game, running on a single multicore machine. They ran into critical deadlock and slowdown issues due to lock contentions.

Adopted Solution: Students were forced to reengineer the whole game and completely understand the interactions between components. By removing redundant locks and following a critical section pattern for every component, they achieved a correct and fast implementation. They drew a schema of interaction of various components, mainly in order to understand the activity of workers at any given moment, and the resource usage and sharing.

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

Adopted Solution: They ran tests to identify which branches were taken most often and adapted the code accordingly.

Problem: Expert developers were implementing an equivalent of an OpenMP "parallel for" loop. They were building a library for other developers to use and ran into a problem where their implementation did not perform as well as expected in production, far below the theoretical performance. They had been very focused on the implementation and theoretical performance and did not consider the context, in this case, the influence of the operating system which negatively impacted on the performance.

Adopted Solution: They adopted a clever solution, where workers would steal work from other workers (work-stealing). This allowed them to have an implementation which was more dynamic in nature and could cope with the uncertainty introduced by the operating system.

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

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

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

"Optimal" solutions

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

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

Picking an adequate orchestration model is not an easy task as it depends on various constraints. We have previously illustrated some of such constraints, including the correctness or performance, but an orchestration model, as opposed to a design pattern, can also have more general constraints, such

as usability. To illustrate this, consider the following quote from an expert developer who worked on a "parallel for" loop abstraction, from the third example.

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

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

Summary

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

DISCUSSION

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

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

This observation leads us to consider a range of implications. Firstly, tools that support parallel programming activity should consider both correctness and performance issues at the same time. This is different from traditional programming where we can see two major types of tools: profilers which are used for measuring and analysing the performance of a system, and debuggers used for diagnosing correctness problems. Secondly, this observation also means that it might be possible to apply research on debugging in software engineering to parallel performance problems, such as the information foraging perspective [23, 35].

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

Therefore we suggest features supporting incremental running and testing of programs should be included in software performance analytics systems. It has been previously noted that such features help developers to be more productive during the debugging process [14, 13]. Hence, enhancing software performance analytics systems with immediate feedback and exploratory search could result in more productive performance analysis. As an example, consider a visualisation system of performance data that supports an exploratory search process, allowing developers to localise performance bottlenecks and hence narrow the range of possible causes. Moreover, such a visual analytics tool could aid program comprehension and help reduce bugs [12]. While this strategy is applicable to more traditional programming tasks, parallel programming introduces more parameters which might be varied, such as conducting multiple runs with different numbers of threads, or changing processor affinity.

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

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

Consider the probe effect - Developers are concerned about the probe effect. Most tools rely on code instrumentation, and hence alter the behaviour of the program. In some cases, this leads to developers avoiding using any tools and relying solely on intuition. The probe effect also makes developers avoid using performance analysis tools for continuous monitoring of production systems. Once the system is deployed, developers desire maximum performance of the system and anything that might slow down or impair a deployed program is avoided. This makes it difficult to measure the performance, capture and diagnose performance issues on production servers, when it occurs. It is especially problematic with non-deterministic issues which may occur only rarely.

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

Make use of orchestration models - During the interviews, developers discussed a range of design patterns and architectures they used throughout their programs. An interesting design direction would hence be to leverage orchestration models in the design of software performance analysis tools. Developers seek to pick a suitable orchestration model for a particular scope. It can be a design pattern, a way to split an array in a function for a parallel loop, or even a cluster configuration. A good tool should help them in this endeavour of "thinking parallel", which is also the number one point on a recent list of challenges in parallel computing [26].

One can think of numerous ways to take advantage of orchestration models. One direction would be to design visualisations which are specific to particular design patterns, and which emphasise features of interest within these, illustrating the performance data with respect to the pattern. One example of such a visualisation can be found in Sutter [15], in which scalability of a given algorithm is charted for a range of different numbers of producer threads and consumer threads, within a producer-consumer orchestration model.

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

CONCLUSIONS

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

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

The study can help to inform the discussion on potential tool support for developers, and particularly the design of performance analysis tools. Future work will involve the design, implementation and evaluation of a performance tool, building on insights and requirements emerging from the study. There is also scope for further investigation of parallel programming practices using other methodologies.

Acknowledgements

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

REFERENCES

1. LJ Ball and TC Ormerod. Structured and opportunistic processing in design: A critical discussion. *International Journal of Human-Computer Studies*, 1995.
2. J. Bonar and B. Liffick. A visual programming language for novices. Technical report, 1987.
3. J. Bonar and E. Soloway. Preprogramming Knowledge: A Major Source of Misconceptions in Novice Programmers. *Human-Computer Interaction*, 1(2):133–161, 1985.
4. BD Boulay, T O'Shea, and J Monk. The black box inside the glass box: Presenting computing concepts to novices. *International Journal of Human-Computer Studies*, 51(2):265–277, 1999.
5. B Cantrill, MW Shapiro, and AH Leventhal. Dynamic Instrumentation of Production Systems. *USENIX Annual Technical Conference*, 2004.
6. T Casavant. Tools and Methods for Visualization of Parallel Systems and Computations Guest Editors Introduction, June 1993.
7. J. Gould; P. Drongowski. An Exploratory Study of Computer Program Debugging. *Human Factors*, 16:258–277, 1974.
8. R. Eccles and D. Stacey. Understanding the Parallel Programmer. *20th International Symposium on High-Performance Computing*, pages 12–12, 2006.
9. V. Fix, S. Wiedenbeck, and J. Scholtz. Mental representations of programs by novices and experts. In *Proc. INTERACT '93/ACM CHI '93*, pages 74–79, 1993.
10. T. Fritz and G.C. Murphy. Using information fragments to answer the questions developers ask. *Proc. ACM/IEEE ICSE 2010*, 1:175, 2010.
11. J. Gait. A probe effect in concurrent programs. *Software, practice & experience*, 16(3):225–233, 1986.
12. D.R. Goldenson and B.J. Wang. Use of Structure Editing Tools by Novice Programmers. In *Empirical Studies of Programming: Fourth Work*, pages 99–120, 1991.
13. T R G Green and M Petre. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996.

14. L. Gugerty and G Olson. Debugging by skilled and novice programmers. In *Proc. ACM CHI '86*, number April, pages 171–174, 1986.

15. H. Sutter. Understanding Parallel Performance. *Dr. Dobb's Journal*, 2008.

16. J.E. Hannay, C. MacLeod, J. Singer, H.P. Langtangen, D. Pfahl, and G. Wilson. How do scientists develop and use scientific software? *2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, pages 1–8, May 2009.

17. MT Heath. Visualizing the performance of parallel programs. *IEEE Software*, pages 29–39, 1991.

18. Lorin Hochstein and Jeff Carver. Parallel programmer productivity: A case study of novice parallel programmers. *High Performance Networking and Computing*, pages 1–9, 2005.

19. G. Kammer, P.and Bolcer, R. Taylor, A. Hitomi, and M. Bergman. Techniques for Supporting Dynamic and Adaptive Workflow. *Computer Supported Cooperative Work*, 9(3-4):269–292, 2000.

20. AJ Ko and BA Myers. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering*, 32(12):971–987, 2006.

21. J Koenemann and SP Robertson. Expert problem solving strategies for program comprehension. *Proc ACM CHI '91*, pages 125–130, 1991.

22. T.D. LaToza and B.A. Myers. Hard to answer questions about code. In *"Second Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU'2010) at SPLASH/Onward!"*, 2010.

23. J. Lawrance, C. Bogart, M. Burnett, and R. Bellamy. How people debug, revisited: An information foraging theory perspective. *IEEE Transactions on Software Engineering*, 39(2):197–215, 2009.

24. CH LeDoux and DS Parker Jr. Saving Traces for ADA Debugging. *ACM SIGAda Ada Letters*, 1985.

25. M. Luff. Empirically investigating parallel programming paradigms: A null result. *Usability of Programming Languages and Tools*, (October), 2009.

26. T. Mattson and M. Wrinn. Parallel programming: can we PLEASE get it right this time? *Proc. 45th Annual Design Automation Conference*, pages 7–11, 2008.

27. R.E. Mayer. The Psychology of How Novices Learn Computer Programming. *ACM Computing Surveys*, 13(1):121–141, January 1981.

28. R. McCauley, S. Fitzgerald, G. Lewandowski, L. Murphy, B. Simon, L. Thomas, and C. Zander. Debugging: a review of the literature from an educational perspective. *Computer Science Education*, 18(2):67–92, June 2008.

29. P.E. McKenney, M. Gupta, M.M. Michael, Phil Howard, Joshua Triplett, and Jonathan Walpole. Is parallel programming hard, and if so, why? *Control*, 2002.

30. B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn parallel performance measurement tool. *Computer*, 28(11):37–46, 1995.

31. MJ Muller and S. Kogan. Grounded theory method in HCI and CSCW. *Cambridge: IBM Center for Social Software*, pages 1–46, 2010.

32. CM Pancake and S Utter. Models for visualization in parallel debuggers. In *Proc. ACM/IEEE conference on Supercomputing*, pages 627–636, 1989.

33. J Pane and B Myers. Usability issues in the design of novice programming systems. Technical Report August, 1996.

34. D. Piorkowski, S. Fleming, I. Kwan, M. Burnett, C. Scaffidi, R. Bellamy, and J. Jordahl. The whats and hows of programmers' foraging diets. *Proc. ACM CHI 2013*, pages 3063–3072, 2013.

35. P. Pirolli and S. Card. Information foraging in information access environments. *Proc. CHI '95*, pages 51–58, 1995.

36. P. Prabhu, Y. Zhang, S. Ghosh, D.I. August, J. Huang, S. Beard, H. Kim, T. Oh, T.B. Jablin, N.P. Johnson, M. Zoufaly, A. Raman, F. Liu, and D. Walker. A survey of the practice of computational science. *State of the Practice Reports on - SC '11*, page 1, 2011.

37. S. Fleming. *Successful Strategies for Debugging Concurrent Software: An Empirical Investigation*. PhD thesis, 2009.

38. C. Sadowski and A. Shewmaker. The Last Mile : Parallel Programming and Usability. *FOSER*, 2010.

39. J. Sillito, G.C. Murphy, and K. De Volder. Questions programmers ask during software evolution tasks. In *SIGSOFT'06/FSE-14: Proceedings of the 13th ACM SIGSOFT and 14th international symposium on Foundations of Software Engineering*, 2006.

40. S Siu, M De Simone, D Goswami, and A Singh. Design Patterns for Parallel Programming. *PDPTA*, 1996.

41. DC Smith, A Cypher, and J Spohrer. KidSim: programming agents without a programming language. *Communications of the ACM*, 37(7):54–67, June 1994.

42. A.L Strauss and J. Corbin. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. Sage, 2nd edition, 1998.

43. I. Vessey. Expertise in Debugging Computer Programs: An Analysis of the Content of Verbal Protocols. *IEEE Transactions on Systems, Man, and Cybernetics*, 16(5):621–637, September 1986.