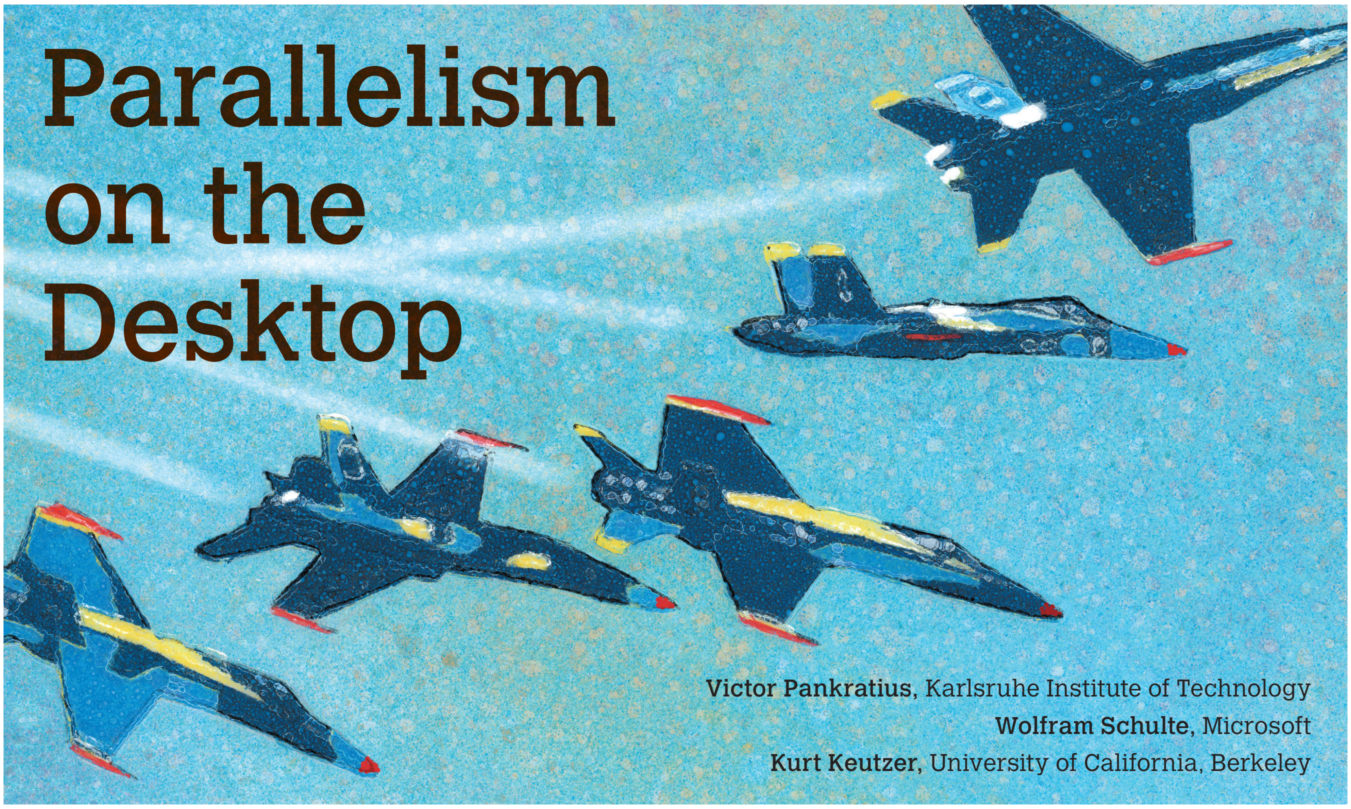
Parallelism on the Desktop



Victor Pankratius, Karlsruhe Institute of Technology
Wolfram Schulte, Microsoft
Kurt Keutzer, University of California, Berkeley

THE COMPUTER INDUSTRY IS EXPERIENCING a major shift: improved single-processor performance via higher clock rates has reached its technical limits due to overheating. Fortunately, Moore's law still holds, so chip makers use transistors to boost performance through parallelism. Modern chips consist of multiple microprocessors (also called cores), buses, and cache memory on the same chip. As of this writing, desktop processors already have up to six cores, but this number will likely increase. Server processors such as Intel's Single Chip Cloud Computer demonstrate that it's possible to integrate 48 general-purpose processors on just 567 mm^2. The record, however, is held by manycore graphics cards with processors that have hundreds of specialized cores, simultaneously executing tens of thousands of hardware threads. In fact, multicore general-purpose processors and manycore graphic processors are the de facto standard for every modern desktop or laptop computer.

Exploiting the full hardware potential of these processors requires parallel programming. Thus, a large number of developers need to parallelize desktop applications, including browsers, business applications, media processing, and other domain-specific applications. This is likely to result in the largest rewrite of software in the history of the desktop. To be successful, systematic engineering principles must be applied to parallelize these applications and environments. In light of these developments, we're pleased to present this special issue on programming methods, tools, and libraries for parallelizing desktop applications.

The Rise of Multicore and Manycore Processors

Let's revisit history for a moment to understand why multicore and manycore processors are here to stay and why software must catch up. Between 1986 and 2004, microprocessor manufacturers used transistors to develop more complex pipelines that increased clock rates even faster than semiconductor scaling alone would allow. The increase in silicon real estate also allowed for more sophisticated processor architectures with speculative execution. All of these factors allowed the desktop programmer to simply write sequential code that principally executed in a single thread of control and still see dramatic performance improvements. But after nearly 20 years, this run of good luck came to an abrupt halt as high clock frequencies pushed power dissipation beyond the limits of economical integrated circuit packages.

This upper limit on power budgets back-propagated to constrain clock frequencies and limit speculative execution

0740-7459/11/$26.00 © 2011 IEEE

in hardware. If software applications were going to continue to see improvements in performance, they needed to apply a radically different strategy. The alternative was to cap clock frequencies and use transistors to implement multiple copies of a single power-efficient microprocessor architecture. In other words, parallelism began to augment and supplant pipelining as a strategy for improving performance. Although parallelism is a natural solution from the standpoint of hardware design, its demands on software developers are quite different. Pipelining delivered significant performance improvements with no demand on software developers to change their applications. In contrast, the development of parallel software asked programmers to completely rethink how to organize their programs so that multiple cores could operate on them simultaneously.

Why Is Parallelism on the Desktop Different?

Servers in the cloud have been able to easily exploit that native parallelism for some time. For instance, serving static files on a Web server requires little or no communication of results between different Web requests, thus this task is easy to parallelize. Because of the relative independence of the tasks, such applications tend to be called "embarrassingly parallel." To assist in processing huge datasets in the cloud, Google introduced the MapReduce framework. Using this framework, Web queries—such as Web index construction, statistical machine translation, image stitching, and news aggregation—can often be *Mapped* simultaneously onto disjoint sets of data distributed throughout the cloud. The same MapReduce framework can then *Reduce* or aggregate the results of the mapped function to deliver, for example, the answer to the query. Thus parallelizing applications that are amenable to implementation in such a framework is quite easy.

At the other extreme, embedded and mobile systems use modest numbers of heterogeneous processors to execute several applications simultaneously—for example, in an automobile or even a mobile phone. At this extreme, energy efficiency plays a major role. Parallel software is typically created from scratch, with the high-level specification of the overall application providing the blueprint for creating parallel tasks.

On the desktop, we face a very different challenge. We must migrate large bodies of existing code written for execution on a single processor to either multicore or manycore computers. To gain any significant performance improvement from this migration, we have to expose the application's intrinsic concurrency as different threads of control. However, as soon as we introduce different threads, program execution is no longer deterministic: the operating system's thread scheduler starts and stops threads running on different cores using its own policy. The resulting thread schedules are inherently unpredictable.

Different threads of control often need access to shared resources. If the shared resource is mutable, programmers must introduce exclusion mechanisms, such as locks in C++ or synchronized methods in Java, to prevent race conditions (where two or more threads change the resource at the same time and introduce an inconsistent state). The protocol of acquiring and releasing resources is often nontrivial: programmers have to decide which parts of the shared state must be protected and for which periods of execution. Improper usage of exclusion mechanisms can lead to contention (when too many threads wait to acquire the same resource), deadlock (when two or more threads are waiting for each other to release a resource), or starvation (when a thread is perpetually denied access to a resource). To avoid these problems, programmers need systematic engineering guidelines and the support of software engineering tools.

In This Issue

In response to these timely challenges, this special issue introduces multicore and manycore software engineering for desktop applications. It presents practically relevant methods, libraries, and tools, as well as exemplary parallelization experiences. The authors also sketch the frontiers of research, painting a picture of the developments that can be expected over the coming years.

In "A Refactoring Approach to Parallelism," Danny Dig addresses an important problem for practitioners who are tasked to produce parallel code out of existing sequential code. Specifically, he introduces different Java refactorings for thread safety, throughput, and scalability, and presents a semiautomated toolset that helps software engineers finish their work more quickly.

Wooyoung Kim and Michael Voss provide in "Multicore Desktop Programming with Intel Threading Building Blocks" an excellent example of a widely used parallel library and explain how to use it. Parallel libraries like this one are extremely helpful to reduce the development time of real-life parallel desktop applications. In addition, the authors also explain the library's internals and design rationales.

Nasser Giacaman and Oliver Sinnen present a practitioner-oriented view on the parallelization of desktop applications with graphical user interfaces in "Object-Oriented Parallelization of Java Desktop Programs." The article elaborates on a new context for parallelization of applications with irregularly structured computations, user interaction, and short runtimes. Giacaman and Sinnen introduce a language called ParaTask, which includes parallel patterns that are useful for the development of desktop applications. The article also shows a new perspective on novel parallelization opportunities on the desktop that differ from classical high-performance computing.

Chi-Keung Luk, Ryan Newton, William Hasenplaugh, Mark Hampton, and Geoff Lowney show how to use an autotuning approach to optimize the performance of multithreaded programs in "Synergetic Approach to Throughput Computing on x86-Based Multicore Desktops." Specifically, they automatically generate many program variants and let a system empirically try out each variant on a target multicore machine until an optimum is found. Their XTune approach saves developers the time involved in manually varying parameters such as ones affecting data size to fit in caches, number of threads, or scheduling policy.

Finally, Frank Feinbube, Peter Troeger, and Andreas Polze show a novel frontier for parallelization on the desktop—the simultaneous usage of multicore processors and graphics processors—in "Joint Forces: From Multithreaded Programming to GPU Programming." The authors give an overview of the technical details for getting started with the development of a new generation of heterogeneous parallel applications. They also discuss best practices on algorithm design, memory transfer and data sizes, control-flow handling, and computational precision.

The importance of concurrency is universally acknowledged. On 8 July 2008, for example, Anders Heilsberg, designer of C#, said, "We have been ignoring concurrency because we could … now we can't … it is a damn hard problem…"(http://channel9.msdn.com/blogs/charles/c-40-meet-the-design-team). Read the articles in this special issue, and see how you can exploit concurrency on the desktop today.

ABOUT THE AUTHORS

VICTOR PANKRATIUS heads the Multicore Software Engineering investigator group at the Karlsruhe Institute of Technology, Germany. He also serves as the elected chairman of the Software Engineering for Parallel Systems (SEPARS) international working group. Pankratius's current research concentrates on how to make parallel programming easier, and his work covers a range of research topics including empirical studies, auto-tuning, language design, and debugging. He has a Dr.rer.pol. with distinction from the University of Karlsruhe, and is a member of the IEEE Computer Society, the ACM, HiPEAC, and the German Computer Science Society. Contact him via www.victorpankratius.com.

WOLFRAM SCHULTE is a principal researcher and the founding manager of Microsoft's Research in Software Engineering (RiSE) team in Redmond, Washington. His research concentrates on improving software development productivity by providing better methods, languages, and tools for describing, developing, analyzing, testing, and executing software. He co-designed Microsoft's Task Parallel Library and recently worked on a Verifier for Concurrent C. He has a PhD from TU Berlin and a state doctorate from the University of Ulm.

KURT KEUTZER is a principal investigator at the University of California, Berkeley's, Universal Parallel Computing Research Center, where he focuses on patterns and frameworks for efficient parallel programming. He's also a professor of electrical engineering and computer science at UCB. Keutzer has published six books and more than 200 refereed articles. He has a PhD in computer science from Indiana University and is a fellow of IEEE.