Ubiquity Symposium

The Multicore Transformation

The Multicore Transformation: Closing Statement

by Walter Tichy

Editor's Introduction

Multicore CPUs and GPUs have brought parallel computation within reach of any programmer. How can we put the performance potential of these machines to good use? The contributors of the symposium suggest a number of approaches, among them algorithm engineering, parallel programming languages, compilers that target both SIMD and MIMD architectures, automatic detection and repair of data races, transactional memory, automated performance tuning, and automatic parallelizers.

The transition from sequential to parallel computing is now perhaps at the half-way point. Parallel programming will eventually become routine, because advances in hardware, software, and programming tools are simplifying the problems of designing and implementing parallel computations.

Ubiquity Symposium

The Multicore Transformation

The Multicore Transformation: Closing Statement

by Walter Tichy

A revolution in programming is underway: The transition to ubiquitous, parallel computers. The contributors to the symposium are in unanimous agreement that we need new techniques to master the new technology, and that the rewards from doing so will be bountiful. Most of them also think that writing parallel software is still too difficult for the average programmer. For example, Wolfram Schulte of Microsoft finds organizing the computation for acceptable performance is more difficult than expressing the parallel algorithm. It is unclear whether disappointing performance (sometimes below that of a sequential implementation) stems from poor language abstractions or poor implementations of these abstractions in hardware and software. From an educational point of view, Keith Cooper of Rice University states: "If all computers are parallel, all courses should deal with parallelism," and one might add: "All programmers should, too."

What Have We Learned?

Not surprisingly, all authors look for performance from multicore machines. Increasing performance is the primary reason for multicores because clock rates are no longer rising and instruction-level parallelism does not scale any further. Additional performance can only be obtained by running several computations simultaneously.

At the same time, multicore graphical processing units (GPUs) are expanding their reach. They now handle the graphics parts of computations with enormous computing power (nearly 3000 cores at the top of the line) and high energy efficiency. These chips work best with applications that use SIMD-style processing. Many applications fit this requirement—they usually perform identical operations on masses of data, a situation that fits SIMD well. Numerical applications, data analytics, gene sequencing, real-time video and stereo processing, learning algorithms,

speech processing, and other applications are being ported to GPUs. Mark Silberstein of Technion notes in the near future, GPUs may no longer need a host processor to get their data, but may access system resources such as files and network connections on their own. Programming GPUs is also becoming simpler, thanks to advances in hardware architecture and libraries. We can now begin to imagine compilers that generate code for both CPUs and GPUs from the same source program. Such compilers would enable massive performance improvements.

Synchronization bugs such as data races and deadlocks make every programmer cringe. These defects are can be extremely hard to find, as they may not manifest themselves in thousands of tests. Synchronization implemented with traditional locks gives only a partial solution to this problem. It is difficult to prove that a set of locks in a complex program actually prevents races. High-contention locks can easily degrade performance. Programs using locks do not compose easily. Maurice Herlihy of Brown University proposes replacing locks with transactions. He has adapted the well-established concept of database transactions to multicore systems and shared main memory. By design, competing transactions appear to run in sequence, when in fact they may overlap as long as they do not conflict. Transactions are inherently easier to use than locks and semaphores. In fact, many problems solve cleanly with transactions, but not with locks. Transactions have fewer contention problems. Programmers merely need to wrap computational steps that operate on shared data in transactions and synchronization is automatic. Chip manufacturers have responded and have begun provide hardware support.

Another exciting area is automated performance tuning, or auto-tuning for short. In order to get decent parallel performance, programmers must juggle numerous parameters, such as the number of threads to use (to avoid overloading the memory system), which pipeline stages to replicate (to eliminate bottlenecks), which pipeline stages to fuse (to reduce the overhead of buffering), which code optimization to use (to get the fastest code), or which of several alternative algorithms to select (which may be data-dependent). One may also want to try out alternative parallelization strategies or choose the best processor for the job (CPU, GPU, or a mixture). Since satisfactory performance prediction models are lacking, programmers need to try the alternatives, which means searching through a huge parameter space. To relieve the programmer of this work, auto-tuning systematically varies the parameters and measures performance for each setting, searching for a near-optimal configuration on a given platform. These measurement runs are executed before production runs begin. For this to work, programmers must prepare their programs such that tuning-knobs are available for the tuner to set. Thomas Fahringer of the University of Innsbruck notes online auto-tuning can do what programmers cannot: Change some of the parameters at runtime, to adapt an application to

changing characteristics of the data or the load on the system. Auto-tuning might solve the problem of portable performance, i.e., achieving acceptable performance on a range of platforms. It can also optimize for other criteria, such as energy consumption, or a combination, as Fahringer explains. If parallel applications are delivered via the web, automatic adaptation to target platforms will be essential. Interestingly, programmers of the '50s and '60s faced a similar problem: They had to worry about overlays, arranging subroutines on pages, prefetching programs and data, and other performance-critical aspects. Virtual memory and adaptive page replacement algorithms solved those tuning problems. The general notion of search-based, automatic optimization is now being applied successfully to multicore software.

Parallel algorithms and parallel architectures go hand in hand. Kenneth Ross of Columbia University demonstrates, as in traditional programming, there is a range of algorithmic choices for a problem; for example, there are many ways to do data aggregation in databases. Peter Sanders of KIT elucidates how algorithm engineering extends sequential algorithms analysis for parallel algorithms. Algorithm engineering features a repeating cycle in which an algorithm is designed, performance tested, and modified in response to the test results. This type of feedback is especially important for parallel computers, for which we have no good predictive models. It should be interesting to see whether algorithm engineering can use auto-tuning to automate part of the cycle.

The symposium covered a lot of ground, though not nearly all the topics. An example is testing and debugging of parallel applications. What if we could identify and even fix (!) race conditions automatically? Research in this area also seeks to identify correlated variables that should be protected as a unit. Once a race is found, locks or transactions could be added automatically to eliminate the race. Another important example is transforming sequential applications into parallel ones. Past research concentrated on parallelizing loops in numerical applications; current research is searching for program patterns that can be transformed into parallel ones, such as pipelines, master-worker structures, parallel divide-and-conquer, and others. Rather than relying on static analysis alone, parallelizers are now using profiling to identify parts of the program that are worth tackling as well as choosing an appropriate pattern.

When Will the Multicore Transformation Complete?

Normally, it takes 20 to 25 years for a new technology to be widely adopted. If we peg the start of multicores at 2005, then we have come not even halfway, even though all computers are already parallel. Can we speed up this process? It is unfortunate that researchers and early

adopters of parallel processing tended to emphasize the problems they encountered with parallel systems. In this environment, newcomers easily acquire a fear of parallel programming. For example, if most of our talk is about unavoidable synchronization bugs, quality-conscious developers will avoid parallelism. The antidote for the pessimism is to also discuss the successes, which are becoming more numerous and more impressive. In my own experience, writing parallel programs is actually not that difficult (an opinion that Schulte shares) and is sometimes easier than sequential programming. The most difficult part is trying out different combinations of the parameters to get the most performance from the system. Programmer must master a few new areas including parallel algorithms, parallel patterns, and synchronization; but they already know all the other programming concepts such as variables, data types, data structures, classes, control constructs, recursion, and I/O. The additional material they must learn for parallelism is manageable. I tell my students they need not fear parallelism—after all, multicore chips are nothing more than replicated Von-Neumann machines. Mastering them will be a job guarantee in the future.

About the Author

Walter Tichy is professor of software engineering at Karlsruhe Institute of Technology (formerly University of Karlsruhe) and a director of the Forschungszentrum Informatik, a technology transfer institute. He is both a Distinguished Scientist and a Fellow of the ACM, and an associate editor of *ACM Ubiquity* and *IEEE Transactions on Software Engineering*. He earned M.S. and Ph.D. degrees from Carnegie Mellon University. He received the Intel Award for the Advancement of Parallel Computing, the ACM Sigsoft Impact Paper Award, and the IEEE Most Influential Paper Award, among others.

DOI: 10.1145/2618409