Ubiquity Symposium

The Multicore Transformation

Making Effective Use of Multicore Systems: A Software Perspective

by Keith D. Cooper

Editor's Introduction

Multicore processors dominate the commercial marketplace, with the consequence that almost all computers are now parallel computers. To take maximum advantage of multicore chips, applications and systems should take advantage of that parallelism. As of today, a small fraction of applications do. To improve that situation and to capitalize fully on the power of multicore systems, we need to adopt programming models, parallel algorithms, and programming languages that are appropriate for the multicore world, and to integrate these ideas and tools into the courses that educate the next generation of computer scientists.

Ubiquity Symposium

The Multicore Transformation

Making Effective Use of Multicore Systems: A Software Perspective

by Keith D. Cooper

The widespread adoption of multicore processors poses several critical challenges for the practice of computer science—challenges in research, in software development, and in education. For roughly two decades, unicore processor performance increased steadily, driven by the twin forces of increasing clock frequency and increasing gate counts. As devices reached practical limits on processor clock speed, the steady march of unicore improvements became unsustainable, so processor architects and manufacturers shifted their designs in the direction of multicore—chips that achieve generation-to-generation performance improvements by building small-scale multiprocessors on a single chip. In the multicore paradigm, performance increases derive from increasing the number of cores and from introducing specialized cores.

The shift from unicore processors to multicore processors fundamentally changed the relationship between application structure and performance. In the unicore paradigm, most applications would run faster with each new generation of chips, as processor clock speed increased. On multicore systems, increased performance comes from inter-core parallelism; to capitalize on the available computing power, an application must be structured so that it can concurrently execute independent instruction streams on separate cores. An application with abundant parallelism will run quickly. An application with little or no parallelism will use just one core and, thus, see little or no benefit from the presence of multiple cores. Unfortunately, many applications fall into this second category.

An ideal multicore application would execute correctly and efficiently across a wide variety of processor configurations. It would adjust its behavior at runtime to match the available resources—both processor cores and dynamic load conditions. Building such applications is difficult. It requires an understanding of parallel computation. It requires tools that let the programmer express and expose parallelism. It requires runtime systems that measure and report dynamic load conditions, and that react in predictable ways to changes in load. These

tools and technologies should be easily usable by a broad audience of programmers; to meet that goal we must improve both our tools and our training.

Over the last 30 years, parallel systems have proven effective in several areas.

- In high-performance computing, history has shown that experts can build and tune specialized applications for systems that range from a handful of parallel threads to thousands or millions of threads. However, the small number of available experts limits the number of truly scalable applications.
- In computer graphics, specialized graphics processing units (GPUs) use parallel hardware to achieve high performance. Software interfaces were introduced to simplify GPU programming. These interfaces have made GPU performance accessible to some applications beyond traditional problems in graphics. In general, problems that fit the underlying hardware model perform well; problems that fit the hardware model poorly do not.
- In end-user code, many applications use a small fixed number of threads, for example, to operate a graphical user interface. These small-scale codes with fixed thread counts achieve performance improvements on small multicore systems. However, once the number of cores exceeds the number of threads, they see little or no benefit from additional cores.

The history of parallelism in more general applications software is mixed, at best. Many, if not most, applications are written for a single-thread (or unicore) execution model. These applications will see little or no performance increase from multicore systems, despite two or three decades of research on automatic parallelization of legacy sequential applications.

Challenges

Multicore processors will need a new generation of applications if they are to realize their full performance potential—applications designed and built specifically to execute on multicore systems. These new applications must address three fundamental challenges posed by multicore systems.

Scalable parallelism. Because multicores derive their power from inter‑core parallelism, a successful multicore application must expose enough parallelism to keep the cores busy.

Furthermore, the increase in useful work should increase directly with the number of cores that the application employs.[1]

Resource sharing. For economy, multicores share expensive resources, such as upper-level caches, between the cores. Such sharing makes the fraction of that resource available to each runtime thread dependent on the workloads of other threads and cores. Applications may need to adjust their behavior accordingly.

Heterogeneity. To an increasing extent, multicores are heterogeneous systems with a variety of different kinds of cores.[2] Today, the tools to compile, build, and run programs do not handle heterogeneity well. Toolchains for heterogeneous multicore systems must make it easier to assemble and run those applications.

To capitalize fully on the power of multicore computing, application programmers need tools and techniques to meet each of these challenges. Effective parallel programming must become both easy and commonplace. Programmers must understand parallel programming models and parallel algorithms that are appropriate for both the application and the target computer system. They must use programming languages that expose the parallelism in an application and tools that translate that specification into an efficient execution. We, as a community, must develop a deeper set of intuitions about programming these multicore systems and we must convey that knowledge and insight when we train students to program.

Programming Models

The research community has studied parallel programming for more than three decades. During that time, many models have been developed to help us reason about parallel computations and implement them. As Professor Tichy points out, these range from Flynn's architectural models of SIMD and MIMD computation to the PRAM; from cooperating sequential processes, monitors, and critical sections to concurrent collections. Each of these models was intended to help the program designer reason about the behavior of a parallel application.

[1] Of course, some applications have an upper bound to the number of cores that they can profitably employ. Understanding such limits is a critical part of multicore programming.

[2] For example, the A7 processor in the iPad reportedly has two processor cores, a four core GPU and an image-processing engine. Heterogeneity will continue to increase, with accelerator cores, reconfigurable cores, and, perhaps, cores that implement the same ISA with different resources.

Our problem is not a lack of programming models, but rather a lack of experience using them and mapping applications onto them. Until recently, parallel systems were both scarce and expensive relative to unicore systems. As a result, most programming courses and most programming activity focused on sequential programming models that are appropriate for unicore systems. Today, multicore parallel systems are ubiquitous; students use multicore systems to program, to debug, and to text their friends. We must teach those students to write parallel programs that use parallel models of computation. Only then will we develop enough experience with these models to learn which ones work well and which do not, at various scales of parallelism. That understanding, of course, must feed back into our curriculum and trainingreplicated Von-Neumann machines. Mastering them will be a job guarantee in the future.

Parallel Algorithms

Construction of scalable parallel applications will require extensive and pervasive use of parallel algorithms. Amdahl's law suggests that, as parallelism grows, the sequential portion of an application will eventually become the rate-limiting step. Thus, application developers must be well versed in both the theory and practice of parallel algorithms. To make construction of effective multicore applications accessible to non-specialists, we must teach parallel algorithms to programmers where such algorithms are known—for example, numerical linear algebra and sorting. We must also continue to develop novel parallel algorithms for problems where they are not currently known.

Algorithms themselves are not enough. Each parallel algorithm should have a performance model that explains how the algorithm scales and how parallelism trades off against locality and sharing. The models should be predictive; they should be inexpensive to evaluate. Programmers might use these performance models to select appropriate algorithms at design time; an equally important use for them might be as drivers for runtime selection of algorithms that best fit the actual data and available resources. Ideally, these algorithms will be encoded into well-documented, widely available libraries so that programmers can easily bring the best parallel algorithms to bear on routine applications.

Programming Languages and Tools

Multicore systems will present challenges to the entire tool chain and will demand improvements in programming languages, compilers, runtime support, and performance-understanding tools.

Programming models are most useful when they are accompanied by programming languages that implement them. Parallel programming models have been grafted onto existing languages, such as MPI (or OpenMP) onto Fortran, C, and C++, or SIMD computation onto C to produce C*. New languages, such as Cilk, Chapel, Erlang, Fortress, Occam, and X10 have also been invented; each embodies a programming model. Functional languages and constraint languages have also been used to build parallel programs. While each approach has had its successes, none of them has reached the broad community of application designers who should now write parallel programs.

Compilers and runtimes for parallel languages will need to cooperate if they are to map an executing program, efficiently and effectively, onto the available parallel resources. They must distribute the workload and the data across the available cores in a way that keeps the cores busy without overtaxing the memory system. If a program region has more parallelism than the available cores can effectively use, the compiler and runtime must block the computation, run parallel blocks in some sequential order, and handle the semantic end-cases that arise from sequential execution of a parallel construct.[3] If a program region has insufficient parallelism, the compiler and runtime need to recognize what fraction of the resources they should use to maintain efficient execution. If the processor allows, they might shut down unneeded cores to reduce energy consumption. Nested parallel regions, such as a call to a parallel library routine from inside a parallel loop, raise similar problems.

Heterogeneous multicore systems break down a critical assumption that underpins our toolchains. Since the dawn of compiled code, programmers have moved code between different implementations of the same ISA without much concern. When model-specific differences determine the number of available cores or the ISAs that those cores implement, model-specific optimization and runtime adaptation will become critical determiners of application performance. Furthermore, the toolchains will need to manage not only the complexity of multiple resource sets in a single ISA, but also the complexity of multiple ISAs in a

[3] For example, to maintain the read-before-write semantics of many parallel assignment constructs, a block-sequential execution may need additional temporary storage.

single executable image. The tools must create that image and make it work seamlessly and efficiently.

As a final challenge for tools, consider the problem of understanding program behavior on a multicore system. Programmers need tools that monitor and document runtime behavior, summarize it, and present it to the programmer in a palatable form. Imagine an application that uses 1,024 threads, spread across a 64-core chip. To identify either a bug or a performance bottleneck, the programmer may need to identify, isolate, and visualize the behavior of a single thread. (Exascale systems, built from thousands of multicore chips, will pose an even larger version of this problem.) Gathering, filtering, analyzing, and presenting such data will require a new generation of ideas and tools that embody them.

Education

The final impediment to our ability to make effective use of multicore systems lies in our educational system. If, as now seems clear, all computers in the future will be parallel computers, then all programmers must be taught parallel programming. Today, most programmer training, whether in secondary school, at the university level, or in specialized training programs, teaches the student to program sequential algorithms in a sequential programming language. We do not teach new programmers most of what is known about parallel programming systems. We need to change that behavior. All programmers should be taught to write parallel programs in languages with appropriate support for parallelism. Rather than being relegated to one or two specialized courses, parallelism should be integrated across the entire Computer Science curriculum, from introductory programming to parsing, from robotic control to computational computer graphics. If all computers are parallel, all courses should deal with parallelism.

Conclusion

Despite the investment of 50 or more years in research on parallel systems and their applications, the practice of programming parallel multicore systems is in its infancy. Research efforts have produced many of the necessary tools and techniques that will enable widespread application of parallel programming. As a community, we need to move these ideas into widespread practice. As parallel programming becomes commonplace, we will develop better intuitions about how to build parallel applications. As we gain those insights, we must feed

them back into our education and training. Only in this way will multicore systems realize the promise of their potential.

About the Author

Keith D. Cooper is the Doerr Professor in computational engineering at Rice University, as well as Associate Dean for Research of the Brown School of Engineering at Rice. He was chair of Rice's Computer Science Department from 2002 through 2008. He was a founding member of the compiler group at Rice and continues as an active researcher and mentor. He has more than 75 technical publications and has produced 18 PhDs. He is the co-author, with Linda Torczon, of *Engineering a Compiler,* a textbook for undergraduate-level compiler construction courses. He is a Fellow of the ACM.

Dr. Cooper has published work across a broad range of problems that arise in the construction of optimizing compilers. His work includes algorithms in program analysis and optimization, as well as a series of improvements to graph-coloring register allocation. His group did pioneering work on adaptive compilation. His current research interests include lightweight schemes for runtime adaptation and optimization of event-driven programs.

DOI: 10.1145/2618407