



Multicore Computing and Scientific Discovery

JAMES LARUS

DENNIS GANNON

Microsoft Research

IN THE PAST HALF CENTURY, parallel computers, parallel computation, and scientific research have grown up together. Scientists and researchers' insatiable need to perform more and larger computations has long exceeded the capabilities of conventional computers. The only approach that has met this need is parallelism—computing more than one operation simultaneously. At one level, parallelism is simple and easy to put into practice. Building a parallel computer by replicating key operating components such as the arithmetic units or even complete processors is not difficult. But it is far more challenging to build a well-balanced machine that is not stymied by internal bottlenecks. In the end, the principal problem has been software, not hardware. Parallel programs are far more difficult to design, write, debug, and tune than sequential software—which itself is still not a mature, reproducible artifact.

THE EVOLUTION OF PARALLEL COMPUTING

The evolution of successive generations of parallel computing hardware has also forced a constant rethinking of parallel algorithms and software. Early machines such as the IBM Stretch, the Cray I, and the Control Data Cyber series all exposed parallelism as vector operations. The Cray II, Encore, Alliant, and many generations of IBM machines were built with multiple processors that



shared memory. Because it proved so difficult to increase the number of processors while sharing a single memory, designs evolved further into systems in which no memory was shared and processors shared information by passing messages. Beowulf clusters, consisting of racks of standard PCs connected by Ethernet, emerged as an economical approach to supercomputing. Networks improved in latency and bandwidth, and this form of distributed computing now dominates supercomputers. Other systems, such as the Cray multi-threaded platforms, demonstrated that there were different approaches to addressing shared-memory parallelism. While the scientific computing community has struggled with programming each generation of these exotic machines, the mainstream computing world has been totally satisfied with sequential programming on machines where any parallelism is hidden from the programmer deep in the hardware.

In the past few years, parallel computers have entered mainstream computing with the advent of multicore computers. Previously, most computers were sequential and performed a single operation per time step. Moore's Law drove the improvements in semiconductor technology that doubled the transistors on a chip every two years, which increased the clock speed of computers at a similar rate and also allowed for more sophisticated computer implementations. As a result, computer performance grew at roughly 40% per year from the 1970s, a rate that satisfied most software developers and computer users. This steady improvement ended because increased clock speeds require more power, and at approximately 3 GHz, chips reached the limit of economical cooling. Computer chip manufacturers, such as Intel, AMD, IBM, and Sun, shifted to multicore processors that used each Moore's Law generation of transistors to double the number of independent processors on a chip. Each processor ran no faster than its predecessor, and sometimes even slightly slower, but in aggregate, a multicore processor could perform twice the amount of computation as its predecessor.

PARALLEL PROGRAMMING CHALLENGES

This new computer generation rests on the same problematic foundation of software that the scientific community struggled with in its long experience with parallel computers. Most existing general-purpose software is written for sequential computers and will not run any faster on a multicore computer. Exploiting the potential of these machines requires new, parallel software that can break a task into multiple pieces, solve them more or less independently, and assemble the results into a single answer. Finding better ways to produce parallel software is currently

the most pressing problem facing the software development community and is the subject of considerable research and development.

The scientific and engineering communities can both benefit from these urgent efforts and can help inform them. Many parallel programming techniques originated in the scientific community, whose experience has influenced the search for new approaches to programming multicore computers. Future improvements in our ability to program multicore computers will benefit all software developers as the distinction between the leading-edge scientific community and general-purpose computing is erased by the inevitability of parallel computing as the fundamental programming paradigm.

One key problem in parallel programming today is that most of it is conducted at a very low level of abstraction. Programmers must break their code into components that run on specific processors and communicate by writing into shared memory locations or exchanging messages. In many ways, this state of affairs is similar to the early days of computing, when programs were written in assembly languages for a specific computer and had to be rewritten to run on a different machine. In both situations, the problem was not just the lack of reusability of programs, but also that assembly language development was less productive and more error prone than writing programs in higher-level languages.

ADDRESSING THE CHALLENGES

Several lines of research are attempting to raise the level at which parallel programs can be written. The oldest and best-established idea is data parallel programming. In this programming paradigm, an operation or sequence of operations is applied simultaneously to all items in a collection of data. The granularity of the operation can range from adding two numbers in a data parallel addition of two matrices to complex data mining calculations in a map-reduce style computation [1]. The appeal of data parallel computation is that parallelism is mostly hidden from the programmer. Each computation proceeds in isolation from the concurrent computations on other data, and the code specifying the computation is sequential. The developer need not worry about the details of moving data and running computations because they are the responsibility of the runtime system. GPUs (graphics processing units) provide hardware support for this style of programming, and they have recently been extended into GPGPUs (general-purpose GPUs) that perform very high-performance numeric computations.

Unfortunately, data parallelism is not a programming model that works for all



types of problems. Some computations require more communication and coordination. For example, protein folding calculates the forces on all atoms in parallel, but local interactions are computed in a manner different from remote interactions. Other examples of computations that are hard to write as data parallel programs include various forms of adaptive mesh refinement that are used in many modern physics simulations in which local structures, such as clumps of matter or cracks in a material structure, need finer spatial resolution than the rest of the system.

A new idea that has recently attracted considerable research attention is transactional memory (TM), a mechanism for coordinating the sharing of data in a multicore computer. Data sharing is a rich source of programming errors because the developer needs to ensure that a processor that changes the value of data has exclusive access to it. If another processor also tries to access the data, one of the two updates can be lost, and if a processor reads the data too early, it might see an inconsistent value. The most common mechanism for preventing this type of error is a lock, which a program uses to prevent more than one processor from accessing a memory location simultaneously. Locks, unfortunately, are low-level mechanisms that are easily and frequently misused in ways that both allow concurrent access and cause deadlocks that freeze program execution.

TM is a higher-level abstraction that allows the developer to identify a group of program statements that should execute atomically—that is, as if no other part of the program is executing at the same time. So instead of having to acquire locks for all the data that the statements might access, the developer shifts the burden to the runtime system and hardware. TM is a promising idea, but many engineering challenges still stand in the way of its widespread use. Currently, TM is expensive to implement without support in the processors, and its usability and utility in large, real-world codes is as yet undemonstrated. If these issues can be resolved, TM promises to make many aspects of multicore programming far easier and less error prone.

Another new idea is the use of functional programming languages. These languages embody a style of programming that mostly prohibits updates to program state. In other words, in these languages a variable can be given an initial value, but that value cannot be changed. Instead, a new variable is created with the new value. This style of programming is well suited to parallel programming because it eliminates the updates that require synchronization between two processors. Parallel, functional programs generally use mutable state only for communication among parallel processors, and they require locks or TM only for this small, distinct part of their data.

Until recently, only the scientific and engineering communities have struggled with the difficulty of using parallel computers for anything other than the most embarrassingly parallel tasks. The advent of multicore processors has changed this situation and has turned parallel programming into a major challenge for all software developers. The new ideas and programming tools developed for mainstream programs will likely also benefit the technical community and provide it with new means to take better advantage of the continually increasing power of multicore processors.

REFERENCES

- [1] D. Gannon and D. Reed, "Parallelism and the Cloud," in this volume.