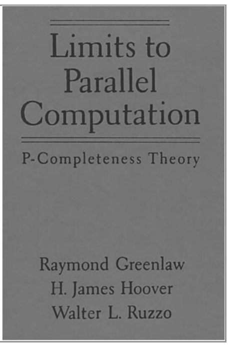
Book Review

Know Your Limits

Igor L. Markov
University of Michigan

Reviewed in this issue

Limits to Parallel Computation: P-Completeness Theory, by Raymond Greenlaw, H. James Hoover, Walter L. Ruzzo. (Oxford University Press, 1995, ISBN-10: 0195085914, ISBN-13: 978-0195085914.)



It is unusual to review a book published 18 years ago. However, some books are ahead of their time, and some prospective readers may have gotten behind the curve. To this end, the development of commercial parallel software is clearly lagging behind initial hopes and promises, perhaps because known limits to parallel computation have been overlooked.

The history of humankind includes several striking technological scenarios that seemed feasible and admitted promising demonstrations, but could not be applied in practice. One example was *perpetual motion*, defined as "motion that continues indefinitely without any external source of energy". The hope was to build a machine doing useful work without being resupplied with fuel. Records of perpetual motion trials date back to the seventeenth century. It took two centuries to formulate the laws of thermodynamics to show why perpetual motion in an isolated system is not possible. A second example is the mythical *philosopher's stone* that transmuted base metals into gold through chemical processes (in fact, published accounts with experimental validation were as respected as modern-day research publications). However, by the late nineteenth century we understood that chemical reactions do not alter chemical elements listed in periodic tables. Both stories show that fundamental limits were discovered, prohibiting initial scenarios. However, this is not how these stories end. Perpetual motion can be successfully emulated by tapping an abundant energy source *while the system remains isolated for practical purposes*, e.g., GPS navigation satellites use solar energy to power their continual transmissions. Another example is nuclear propulsion in ballistic missile submarines that remain submerged and isolated for years. Even the transmutation of cheap metals into gold has been demonstrated in particle accelerators, and platinum-group metals can be commercially extracted from spent nuclear fuel. Once scientists develop an understanding of fundamental limits, engineers circumvent these limits by reformulating the challenge or by other clever workarounds.

Today, the business value in many industries is fueled by computation, just like it was driven by steam engines during the industrial revolution and backed up by precious metals during the tumultuous Middle Ages. The need for faster computation leads to significant investments into computing hardware and software. Just like Chemistry and Physics were developed to study chemical reactions and energy conversion, Computer Science was developed in the last 60 years to study algorithms and computation. In particular, Complexity theory

Digital Object Identifier 10.1109/MDAT.2012.2237133
Date of current version: 11 April 2013.

studies *the limits of computation*, as illustrated by the notion of *NP*-complete problems (a standard textbook is Michael Sipser's "Introduction to the Theory of Computation"). Current consensus is that these problems cannot be solved in worst-case polynomial time without major theoretical breakthroughs, and the knowledge accumulated in the field allows one to quickly evaluate and diagnose purported breakthroughs. Even the least-informed funding agencies would now recognize naïve attempts at solving *NP*-complete problems in polynomial time. On the other hand, the understanding of such limits guided applied algorithm development to identify and exploit useful features of problem instances. An example end-to-end discussion can be found in the DAC 1999 paper "Why is ATPG Easy?" by Prasad, Chong, and Keutzer. Moreover, for optimization problems, the notion of *NP*-hardness can sometimes be circumvented by *approximating* optimal solutions (typical for geometric tasks, such as the Travelling Salesman Problem). As a result, the software and hardware industries have been quite successful in circumventing computational complexity limits in applications ranging from formal verification to large-scale interconnect routing. And chess-playing computers go far beyond NP.

History does not repeat itself, but it often rhymes, as Mark Twain noted. The latest craze in software—parallel computing—has given us hope to turn silicon (predesigned processor cores) into computation without increasing clock speed and power dissipation per core. As top-of-the-line integrated circuits cost more than their weight in gold, the philosopher's stone pales in comparison to the value proposition of turning not base metals, but *sand* into something more expensive than gold. And we now see academics, instigated by U.S. funding agencies left unnamed (to protect the guilty!), claim fantastic parallel speed-ups that do not survive scrutiny. Those who attended the panel on parallel Electronic Design Automation at ICCAD 2011 may recall that I questioned claims of *algorithmic* "superlinear" speed-up (more than *k* times when using *k* processors, for large *k*). If using *k* parallel threads of execution consistently improves single-thread runtime by more than a factor of *k*, then we could just simulate *k* threads by time-slicing a single thread, with a factor-of-*k* slowdown. This yields a better sequential algorithm. Thus, the original comparison was to suboptimal sequential algorithms (using *k* CPU caches can boost memory performance, but only by a constant factor, and not entirely due to parallel algorithms). Other signs that a claimed speed-up is bogus can be subtle and *ad hoc*. For fundamental problems, like Boolean SAT and circuit simulation, that have consistently defied parallelization efforts by sophisticated researchers, a spectacular speed-up (e.g., 220 times claimed at ICCAD 2011 for SAT) better have a convincing and unexpected explanation. Patrick Madden's ASPDAC 2011 paper illustrates how academics often oversimplify the challenge they are studying and ignore best known techniques in their empirical comparisons. David Bailey's SC 1992 paper "Misleading Performance Claims in the Supercomputing Field" and its DAC 2009 reprise suggest that this phenomenon is not new.

The article "Parallel Logic Simulation: Myth or Reality?" in the April 2012 issue of IEEE Computer offers a great exposition of the promise and the failure of parallel functional logic simulation (e.g., evaluating new circuit designs before silicon production). Many people find it *obvious* that Boolean circuit simulation should be easy to parallelize, and academic papers claim such results. But implementing this idea in successful commercial software has been a losing proposition for many years (leaving the market open to expensive *hardware emulators* developed by IBM, EVE, Cadence, Synplicity/Synopsys, and others). The authors of the IEEE Computer article dissect many failed attempts and the obstacles encountered. This is where careful observers may suspect fundamental limits.

Enter the book *Limits to Parallel Computation: P-Completeness Theory* by Greenlaw, Hoover, and Ruzzo. Just like *NP*-complete problems defy worst-case polynomial-time algorithms, *P*-complete problems defy significant speed-ups through parallel computation. The Preface says:

> This book is an introduction to the rapidly growing theory of *P*-completeness—the branch of complexity theory that focuses on identifying the "hardest" problems in the class *P* of problems solvable in polynomial time. *P*-complete problems are of interest because they all appear to lack highly parallel solutions. That is, algorithm designers have failed to find *NC* algorithms, feasible highly parallel solutions that take time polynomial in the logarithm of the

> problem size while using only a polynomial number of processors, for them. Consequently, the promise of parallel computation, namely that applying more processors to a problem can greatly speed its solution, appears to be broken by the entire class of *P*-complete problems.

Just like the well-known book "*Computers and Intractability: A Guide to the Theory of NP-Completeness*" by Garey and Johnson, this book consists of two parts—an introduction to the *P*-completeness theory, and a catalog of *P*-complete problems. It starts with an anecdote about a company that was forced by its competitors to look into parallel platforms and thus developed parallel sorting of n elements using n^2 processors in $O(\log n)$ time. This example is used to motivate key concepts, such as *reductions* and implied limits to parallel computation (as in my earlier argument about the maximal speed-up due to k processors). Similar to the theory of *NP*-completeness, this leads to the notion of *P*-complete problems to which many other problems can be reduced. Thus, when looking for effective parallel solutions to a particular problem, one must first check for reductions to known *P*-complete problems. For example, Linear Programming and Maximum Flow (Problems A.4.3 and A.4.4 in the catalog) are *P*-complete, while Maximum Matching (Problem B.9.7) admits a highly-parallel probabilistic algorithm. This catalog is substantial. It contains multiple variants of problems including restricted versions. For example, Linear Programming with Two Variables per Constraint (Problem B.2.2) and 0-1 Maximum Flow (B.9.6) admit highly-parallel algorithms. Another quote

> Additionally, *P*-completeness theory can guide algorithm designers in cases where a particular function has a highly parallel solution, but certain algorithmic approaches to its computation are not amenable to such solutions. Computing breadth-first level numbers via queue- versus stack-based algorithms is an example (see Chapter 8 for more details).

The two main models of parallel computation are *combinational Boolean circuits* and *shared-memory multi-processors*. Here, a key issue is the efficiency of parallel simulation of a Boolean circuit on a multiprocessor and simulating a multiprocessor by unrolled combinational circuits. Further analysis is based on formal notions of a *computational problem*, *reducibility* and *completeness*. These notions lead to complexity classes, such as *P* (problems solvable in polynomial time) and *NC* (problems solvable by poly-sized circuits of polylogarithmic depth/delay, named "Nick's class" after Nicholas Pippenger). Clearly, *NC* is contained in *P*, but is believed to be smaller than *P* (just like *P* is believed to be smaller than *NP*). Because any problem in *P* can be efficiently reduced (*NC*-reduced) to any *P*-complete problem, finding a *P*-complete problem inside *NC* would contradict $P \neq NC$ (Theorem 3.5.4). So, if you are comfortable interpreting *NP-complete* as "likely not solvable in polynomial time," you should be comfortable interpreting *P-complete* as "likely not executable efficiently in parallel." The prototypical *P*-complete problems are circuit and program simulation.

P-complete problems are "inherently sequential" in the sense that $P = NC$ is unlikely. The reasons have to do with the efficiency of highly parallel simulation and can be summarized as follows: (*i*) generic simulation is slow, regardless of the algorithm used, (*ii*) fast special-case simulation techniques are not general enough, (*iii*) straightforward simulation techniques are provably slow. An additional rule of thumb is that efficiently parallelizable problems can usually be solved in polylog space (in addition to having access to the input). Details can be found in David Johnson's April 1983 Journal of Algorithms column.

Focusing entirely on problems solvable in polynomial time would have excluded tasks in formal verification and logic synthesis, which sometimes venture far beyond *NP*. There is no hope that using polynomially many processors can make even NP-complete problems poly-time solvable, which is consistent with empirical results on Boolean Satisfiability seen today. However, if we are interested in polynomial-time heuristics for problems beyond *P* (which is how most practical work is done), we might first try to parallelize tried-and-true sequential heuristics. To this end, Greenlaw, Hoover, and Ruzzo show that sequential greedy algorithms frequently lead to solutions that are *inherently sequential*, i.e., cannot be duplicated rapidly in parallel, unless $NC = P$. But sometimes equally good solutions can be produced by parallel algorithms. To this end, *P*-complete *algorithms* are discussed through the proxy of tasks to reproduce the output of a particular algorithm. For example, conventional algorithms for Gaussian elimination (with partial pivoting) appear inherently sequential, but other, highly parallel algorithms exist for the same problem. This particular example can be useful in SPICE-like accurate electrical circuit simulation. More generally, such examples motivate pessimism about automatic parallelization by compilers given that compilers generally do not invent entirely new algorithms. Section 9.2 shows some loopholes in dealing with *P*-complete problems and briefly discusses polynomial speed-up in parallelizing special cases of the Circuit Value Problem, Depth-first Search, etc. It also shows how to upper-bound such speed-ups. Another loophole is analogous to that in the *NP*-completeness theory and relies on quick (parallel) approximations that bypass exact algorithms (Chapter 10). Unfortunately, this helps only in rare cases (Section 10.2). Such logic seems more promising in parallelizing approximate sequential solutions to *NP*-complete problems (Section 10.3), such as bin packing, 0-1 knapsack and scheduling—problems inherent in load-balancing on parallel platforms. David Johnson's April 1983 J. Algorithms column reviews such results.

Appendices list problems whose status (*P*-complete or not) is known, as well as open problems. Each problem is defined in a self-contained way, and relevant problem reductions are outlined. Circuit-related *P*-complete tasks include Problem A.1.9 Min-Plus Circuit Value Problem, which can be viewed as a narrow form of Static Timing Analysis with rational values. Problem A.10.1 is a sweeping generalization with real-valued numbers. Other problems of relevance to Computer Engineering include Graph Partitioning, List Scheduling, Linear Programming, Network Flow problems, certain approximations to Max-SAT and Min Set-Cover, and even Two-Layer Channel Routing. Section A.7 lists problems dealing with formal languages (pushdown automata, context-free grammars, etc.), and captures various tasks performed by parsers—a common bottleneck in parallel software. Later sections include Gaussian elimination, various geometry problems (triangulation, convex hulls, etc.), several numerical analysis problems, as well as Lempel-Ziv (LZ) compression. Fortunately, LZ is not an obstacle to parallel I/O because it is applied to small blocks, not to entire files.

Computer engineers have been ignoring fundamental limits to parallel computation for years. For example, the 2006 manifesto "The Landscape of Parallel Computing Research: A View from Berkeley" does not mention complexity limits to parallel algorithms and the concept of *P*-completeness. The Berkeley engineering professors who authored the manifesto represented key parallel applications by "13 dwarfs"—patterns of computation and communication (extending the seven dwarfs defined by Phil Collela). But we are not told that some of these dwarfs are in *NC* (easy to parallelize), some harbor *P*-complete problems (combinational logic, certain graph traversals), some are beyond *P* (branch-and-bound) and some are too broad for generic analysis (dynamic programming). Clearly, this classification is missing an important dimension. Not appreciating computational complexity, computer engineers have been cranking out papers on parallel algorithms for *P*-complete problems without realizing this (students can find those papers and match them to problems in Appendix A). David Bailey's 1991 note "Twelve Ways to Fool the Masses when Giving Performance Results on Parallel Computers" illustrates what many of these papers do. On the other hand, efforts at parallelizing hard problems can be useful, just like ongoing efforts on practical sequential algorithms for *NP*-hard and *NP*-complete problems through approximation and exploiting instance structure. In any case, researchers must clearly understand the fundamental limits they are up against, and a summary of known results in parallel algorithms clarifies what is achievable. For example, most highly parallelizable problems can be solved in polylogarithmic parallel time with a (close-to-) linear number of processors (in terms of input size), but sorting and biconnected components need n^2 processors.

Given that the book under review was published 18 years ago, one may wonder if its conclusions remain valid today. To this end, proven theorems are in no danger of becoming outdated, and the "*P* versus *NC*" challenge remains unresolved, just like its close relative "*P* versus *NP*". However, a few years after the book was published, Ketan Mulmuley proved that the *P*-complete max-flow problem *cannot* be solved in polylogarithmic time using polynomially many processors in the PRAM model under certain assumptions. In case of future breakthroughs on this topic, updates should promptly appear on the

Wikipedia pages for the *NC* and *NP* classes. Modern formal treatment of multicore computing is available in Leslie Vailant's ESA 2008 paper "A Bridging Model for Multi-Core Computing," which discusses algorithms that are optimal for all combinations of machine parameters including the number of cores and the shape of the memory hierarchy. Other practical aspects of parallel algorithms are explored in the 1997 volume "Parallel Algorithms: Third DIMACS Implementation Challenge." For example, a chapter by Papaefthymiou and Rodrigue points out that the Bellman-Ford algorithm runs faster in parallel on dense graphs, but not on sparse graphs.

A major technological change in parallel computing is the increasing dominance of communication over computation. It is not explicitly addressed by the theory of *P*-completeness, but computation costs remain valid lower bounds and determine how much communication is needed. Thus, classical impossibility results and lower bounds on computation can still be trusted, but may be optimistic in practice. To this end, the 1998 IEEE Transactions on Computers paper "Your Favorite Parallel Algorithms Might Not Be as Fast as You Think" by David Fisher accounts for the finite density of processing elements in space, the (low) dimension d of the space in which parallel computation is performed, the finite speed of communication, and the linear growth of communication delay with distance. Neglected in most publications, these four factors limit parallel speed-up to power $(d+1)$. Considering matrix multiplication as an example where exponential speed-up is possible in theory, a two-dimensional computing system (a planar circuit, a modern GPU, etc.) can offer at most a cubic speed-up. Given that the general result is *asymptotic*, it is significant only for large numbers of processing elements that communicate with each other. In particular, for circuits and FPGAs, it limits the benefits of three-dimensional integration to power 4/3 (optimistically assuming a fully isotropic system). For two-dimensional GPUs, at most a cubic speed-up over sequential computation is possible. To this end, a 2012 report by the Oak Ridge Leadership Computing Facility analyzed widely used simulation applications (turbulent combustion; molecular, fluid and plasma dynamics; seismology; atmospheric science; nuclear reactors, etc.). GPU-based speed-ups ranged from 1.4 to 3.3 times for ten applications and 6.1 times for the eleventh (quantum chromodynamics). These mediocre speed-ups likely reflect flaws in prevailing computer organization, where heavy reliance on shared memories dramatically increases communication costs, but alternatives would drastically complicate programming.

As in historical examples at the beginning of the review, the last word on parallel algorithms seems to be with loopholes. Even the core concepts we've discussed exhibit subtle flaws. For example, binary search is obviously in *NC*, but cannot be parallelized efficiently. The 1998 result of David Fisher questions the very relevance of the *NC* class in the physical world, as *no exponential worst-case parallel speed-up can be achieved* in three (or any finite number of) dimensions, even if all interconnects can be routed with smallest possible lengths. Effective loopholes here *hide communication latencies* by connecting slow processor with fast interconnect, exploiting better-than-worst-case data patterns (through pipelining and trading communication for computation), and scaling semiconductor technologies by using repeaters and electric tuning. The most popular loophole is to use an identical interconnect network for all input sets (up to 4 GB or, perhaps, 256 GB) and pretend that interconnect latencies remain constant as problem size grows. But even zero-latency communication would not help with obstacles related to *P*-completeness. In particular, the *P*-completeness of circuit and processor simulation problems explains the difficulties encountered by computer engineers when simulating new hardware designs on parallel systems (here an important loophole is *hardware emulation*). Thus, by warning about important pitfalls, keen understanding of obstacles to parallelism can guide toward more effective solutions, clever ways to reformulate the problem, and applications where speed-up is easier to achieve (data-distributed tasks such as digital cinematography, computational astronomy and Web search). In summary, I am convinced that the book under review can intellectually enrich Computer Engineering research and enhance the level of discourse in the scholarly literature. ■

Acknowledgment

Several colleagues helped improve this book review. Dr. Michael Moffitt and Dr. Raşit Topaloğlu from IBM, and Prof. Scott Aaronson from MIT pointed out several important omissions in early drafts. Prof. Massimo Cafaro from Università del Salento provided technical clarifications on complexity classes (via MathOverflow). Dr. David Johnson of AT&T was helpful in discussing the relative paucity of new developments in the complexity of parallel computation since the book was published. Dr. Grant Martin from Tensilica and Dr. Mehdi Saeedi from the University of Southern Californina helped improve readability.

■ Direct questions and comments about this article to Igor L. Markov, University of Michigan, 2260 Hayward St., Ann Arbor, MI 48109-2121 USA; imarkov@eecs.umich.edu.