DOI:10.1145/2580945

Uzi Vishkin

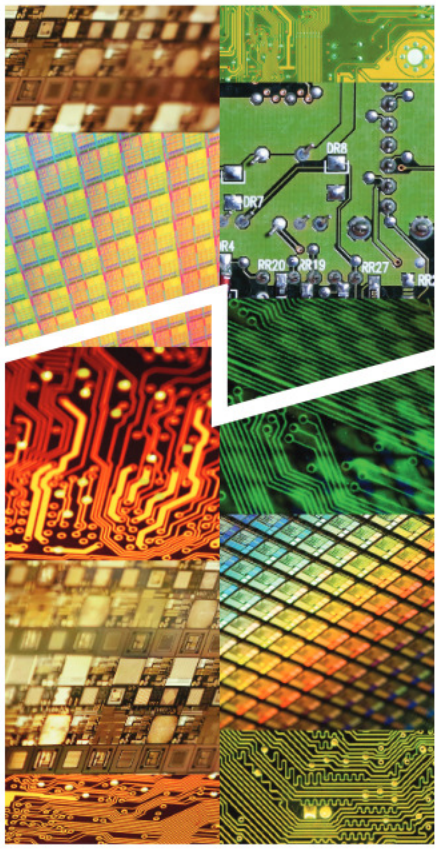
Viewpoint

Is Multicore Hardware for General-Purpose Parallel Processing Broken?

The current generation of general-purpose multicore hardware must be fixed to support more application domains and to allow cost-effective parallel programming.

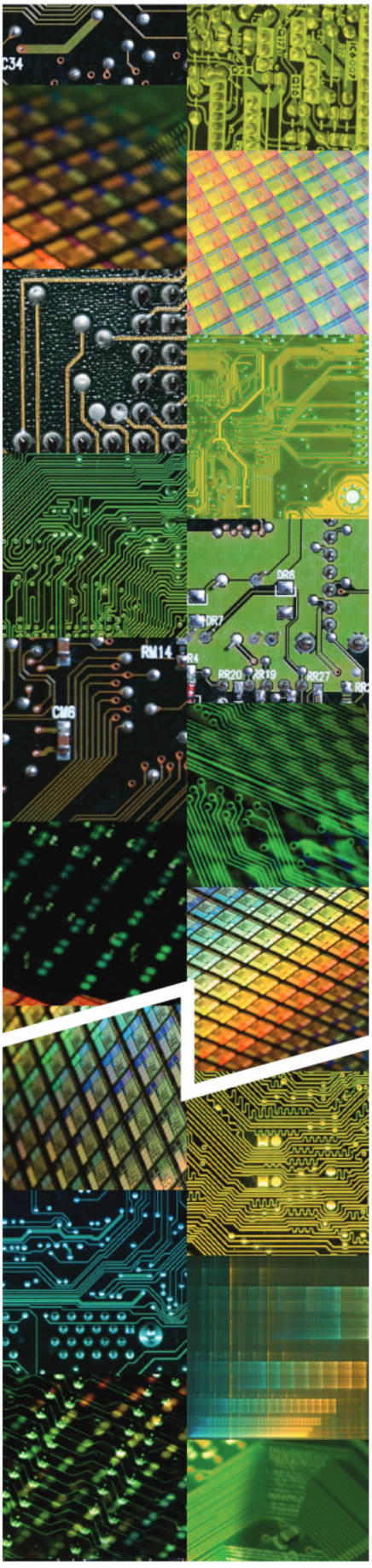
In the recent decade, most opportunities for performance growth in mainstream general-purpose computers have been tied to their exploitation of the increasing number of processor cores. Overall, there is no question that parallel computing has made big strides and is being used on an unprecedented scale within companies like Google and Facebook, for supercomputing applications, and in the form of GPUs. However, this Viewpoint is not about these wonderful accomplishments. A quest for future progress must begin with a critical look at some of the current shortcomings of parallel computing, which is the aim of this Viewpoint. This will hopefully stimulate a constructive discussion on how to best remedy the shortcomings toward what could ideally become a parallel computing golden age.

Current-day parallel architectures allow good speedups on regular programs, such as dense-matrix type programs. However, these architecture are mostly *handicapped on other programs, often called "irregular," or when seeking "strong scaling."* Strong scaling is the ability to translate an increase in the number of cores to faster runtime for problems of fixed input size. Good speedups over the fastest serial algorithm are often feasible only when an algorithm for the problem at hand can be mapped to a highly parallel, rigidly structured program. But, even for regular parallel programming, cost-effective programming remains an issue. The programmer's effort for achieving basic speedups is much higher than for basic serial programming, with some limited exceptions for domain-specific languages where this load is somewhat reduced. It is also worth noting that during the last decade innovation in high-end general-purpose desktop applications has been minimal, perhaps with the exception of computer graphics; this is especially conspicuous in comparison to mobile and Internet applications. Moreover, contrasting 2005 predictions by vendors such as Intel[2] that mainstream processors will have several hundred cores ("paradigm shift to a many-core era") by 2013 with the reality of around a handful; perhaps the reason was that the diminished competition among general-purpose desktop vendors in that timeframe did not contribute to their motivation to take the risks that such a paradigm shift entails. But, does this mean the hardware of these computers is broken?



I believe the answer is yes. Many more application domains could significantly benefit from exploiting irregular parallelism for speedups. A partial list of such domains follows.

- *Bioinformatics.* Genomics involves many graph and other combinatorial problems that are rarely regular.
- *Computer vision.* Only a fraction of the OpenCV library is supported well on graphics processing units (GPUs). Other OpenCV primitives are typically irregular.
- *Scientific computing.* Sparse matri-



ces and problems that require runtime adaptation such as multiscale methods for solving linear equations, or for kinetic modeling of quantum systems.

- Data compression.
- Sparse sensing and recovery in signal processing.
- Electronic design automation (EDA) for both design and simulations.
- Data assimilation; and
- The numerous uses of graph models:
  - Data analytics.
  - Analysis of social networks.
  - Epidemiological networks.
  - Belief propagation.
- *Open-ended problems and programming*. This domain is characterized by the way problems may be posed and computer programs developed rather than by a particular application. Many problems whose initial understanding does not imply clear output, or sometimes even input, definitions lend themselves to irregular programming.

Today's general-purpose multicore hardware does not provide sufficient support for any of these domains. It must be fixed to support more of them, and to allow cost-effective parallel programming. A fix will require changes both to current hardware, and to the overall ecological system comprising them, including programming practice and compilers. The lead questions for such a system are: how should programmers express the parallelism in the algorithms they implement; what will the responsibility of compilers and hardware/software runtime systems be; and how will the hardware execute the eventual ready-to-run parallelism as efficiently as possible. I believe that for irregular problems: programmers are good at seeing parallelism (that is, understand which operations can be done concurrently) well beyond what current and near-future compiler technology can extract from serial code; much of this parallelism is fine-grained; and this parallelism can vary from very scarce (very few operations that can be executed concurrently) to plenty (many such operations), even among steps of the same parallel algorithm. Thus, system designers must "keep their eye on the ball" by viewing the parallelism that programmers provide as perhaps their *most precious resource*. Specifically: encourage programmers to express all the parallelism they see (of course, after facilitating such expression in programming languages), and optimize compiler, runtime systems, and hardware to derive the best performance possible from whatever amount of parallelism programmers provide. In particular, the overheads for translating any amount of parallelism coming from the program to performance must be minimized.

Examples for what could be done include shared (on-chip) caches, low-overhead control mechanisms, high bandwidth, low latency interconnection networks, and flexible data fetch and store mechanisms. Many would be justifiably puzzled by this list and ask: Aren't we already aware of these measures? Aren't they already being implemented? Closer scrutiny would suggest the incorporation of these measures has been greatly diluted by competing considerations (not keeping one's eye on the ball), leading to the current chasm between parallel architectures and irregular parallel algorithms. Several examples of competing objectives that hurt multicore performance on irregular parallel programs follow.

Competing objective: *Improve throughput.* Current multiprocessors consist of tightly coupled processors whose coordination and usage are controlled by a single operating system (OS). This has been a common approach when the objective of the architecture is to maximize the number of (possibly small) jobs to be completed in a given time. (Sometimes this objective is referred to as optimizing throughput.) Being a software system, the OS encounters unnecessarily high overhead for thread management, including: thread initiation, dynamic allocation of threads to hardware, and thread termination. However, had the main objective been low overhead support for irregular parallel programs, we would have seen migration of more of these functions to hardware, especially for *fine-grained programs as many irregular programs are*.

Competing objective: *Maximize peak performance.* Designs of GPUs seem to have been guided by fitting as many functional units as possible within a silicon budget in an attempt to maximize peak performance (for example,

FLOPs). But, effective support of irregular programs, strong scaling, and better sustained (rather than peak) performance is a different objective requiring different choices to be reflected both on the hardware side and on the programmer side. Examples for the hardware side: the data a functional unit needs to process cannot be generally assumed to be available near the functional unit, or when vector functional units are used, data needs to be provided to them both at a high rate and in a structured way, but this cannot be generally assumed, as well. Simple parallel programming is also not compatible with expecting the programmer to work around such data feeds. There are quite a few examples of how GPUs require data to be structured in a very rigid way by the programmer.

Competing objective: *Maximize locality.* The respective roles that caches have come to play in serial and parallel architectures help explain at least part of the problem.

Caches in serial architectures. Serial computing started with a successful general-purpose programming model. As improvement in memory latency started falling behind improvement in serial processor speed during the 1980s, caches emerged as the solution for continued support of the serial programming model. The observation that serial programs tend to reuse data (or nearby addresses of data recently used), also known as the "principle of locality," meant caches could generally mitigate problems with continued support of that model. Thus, while locality has become a major theme for optimizing serial hardware, it was not allowed to interfere with the basic programming model of everyday programming; even when programming for locality was done, it was by relatively few "performance programmers."

Local parallel memories. Parallel computing has never enjoyed a truly successful general-purpose programming model, so there was no sufficient motivation to invest in continued support of one. Parallel processing architectures have been driven since their early days by the coupling of each processor with a considerable local memory component. One key reason, already noted earlier, was the quest for higher peak performance counting FLOPs. This meant maximizing the number of functional units and their nearby memory within a given dollar budget, silicon-area budget, or power budget. I also noted that trading off some of these budgets for improved sustained performance, perhaps at the expense of peak performance, appears to have been a lower priority. Thus, mapping parallel tasks to these local memories has become an enormous liability for the programmer, and one of the main obstacles for extending the outreach of parallel processing beyond regular applications and for making parallel programming simpler.

A quest for future progress must begin with a critical look at some of the current shortcomings of parallel computing.

Competing objective: *Prioritize highly parallel applications.* Current designs seem to expect a very high amount of parallelism to come from applications. I believe that, in general, this view is too optimistic. One lesson of serial computer architecture has been the need to put any general-purpose hardware platform through nontrivial benchmarking stress tests. Downscaling of parallelism on current parallel machines is often a problem for irregular algorithms and problems. For example, in breadth-first search on graphs, some problem instances may provide a large amount of parallelism (for example, random graphs) while other instances do not (for example, high-diameter graphs). Some algorithms operate such that the parallelism in different steps of the algorithm is drastically different. For example, standard max-flow algorithms provide a useful stress test. These max-flow algorithms iterate breadth-first search on graphs of increasing diameter, and therefore their parallelism decreases as the algorithm progresses, failing architectures that can handle well only a high level of parallelism.

Competing objective: *Prioritize energy saving over programmer's productivity.* Approaching the end of the so-called Dennard scaling and the decreasing improvement in power consumption of computers it implies are important concerns.[5] Power consumption is also easier to quantify than programmer's productivity and is closer to the comfort zone of hardware designers. This may explain the often heard sentiment that parallel programmers must take on themselves programming for reducing energy consumption, which found its way into some design decisions. I see two problems with this trend, one is rather concrete and the other is more principled. The concrete problem is that irregular problems make it much more difficult, if not impossible, for programmers to conform with these design decisions. The general problem is that the basic sentiment seems to "go against history." Much of the progress attributed to the Industrial Revolution is due to using more power for reducing human effort. Can future progress in computing performance be based on reversing this trend?

The reader needs to be aware that this approach of questioning vendors' hardware presented here is far from unanimous. In fact, many authors have sought conformity with such hardware, modeling limitations for meeting them in algorithm design. Bulk-synchronous parallelism (BSP),[6] and more recently quite a few communication-avoiding algorithms, such as Ballard et al.,[1] are notable examples for considerable accomplishments regarding regular algorithms. However, the state of the art remains that unless problem instances can be mapped to dense matrix structure, they cannot be solved efficiently, and after many years of parallel algorithms research I do not believe such a fundamental change in reality is feasible.

Interestingly, the chasm between this communication-avoiding school of thought and the position of this Viewpoint is not as large as it may appear. Before explaining why, I point out that historically government investment in parallel computing has been mostly tied to the bleeding edge of large high-performance computers, driven

by large scientific applications. Here, I advocate the advancement of parallel computing in smaller-scale systems for improved runtime, ease and flexibility of programming, and general-purpose applications. If successful, such a small-scale system could also, in turn, provide a better building block (for example, node) for the larger computers. For example, the recent paper by Edwards and Vishkin[3] suggests that such organization of larger computers could double the effective bandwidth between every pair of their nodes, while still using the same interconnection hardware. Many larger computers operate by sending messages from one node to another. For better utilization of bandwidth, the sending node applies a data compression algorithm prior to sending a message, and the receiving node applies a matching decompression algorithm. Using XMT-type computers (see a later reference) at nodes could generally half the size of the compressed message without incurring greater delays for running compression and decompression. This direction offers an interesting middle ground with respect to the communication avoidance school-of-thought. Namely, use a high-bandwidth easy-to-program parallel computing paradigm within the nodes of a large machine and a communication avoidance paradigm among nodes.

I teach upper-division computer engineering majors after they have taken nearly all required programming, data structures, and algorithms courses, many of whom also have some software development and application experience through internships and research. It was interesting for me to hear them out, as they already know a great deal, but have not yet been under much pressure to conform, for example, from employers. They find it difficult to come to terms with the state of the art of parallel programming. As students are presented with the contrast between regular and irregular programs, they point out that a vast majority of the programs they have encountered have been irregular. Understandably, they resent not being able to pursue a similarly flexible style for parallel programming. To them, the *"get-your-hands-dirty" approach to problem solving*, as cultivated by the standard CS curriculum and CS practice, is being diminished by the type of parallel programming mandated by today's parallel hardware. This connects to the previous discussion of open-ended problems and programming. Development of programs for an "ill-defined" problem may be significantly different from one programmer to the other, as well as the program each contributes. Such aspects of individuality and creativity reflect the "soul" of computer science no less than well-understood applications; in particular, these aspects have been instrumental in attracting talent to the CS profession. However, since it is problematic to reflect this mode of open-ended programming in application benchmarks, such programs and their development tend to be underrepresented, if not missing altogether, in stress tests for new parallel processors. This reality has made it too easy to sidestep this domain in traditional, benchmark-driven hardware design.

The challenge for hardware vendors is big. Any paradigm shift, such as that required by the transition to parallelism, is inherently risky. The fact that vendors cannot expect much feedback from application developers before making such hardware available is also not helpful for mitigating the risk. Application software is typically developed for new hardware only after this hardware is available, creating a chicken-egg-problem for vendors. I already noted the diminished competition among desktop and laptop vendors during the last decade has also not added to their motivation. However, the ongoing confluence of mobile and wall-plugged computing is bringing about fierce competition and with it the need to take risks in order to stay competitive. This is good news for the field.

Other good news is the continuing increase in silicon area budgets and the

Any paradigm shift, such as that required by the transition to parallelism, is risky.

advent of 3D-VLSI technology, along with its potential accommodation of greater heterogeneity in hardware, may allow vendors to add new components without removing support for current programming models.

A reviewer of an earlier version of this Viewpoint challenged its basic thrust with the following interesting argument. Since the sole point of parallel computing is performance, every parallel programmer is by definition a performance programmer. Thus, the effort of parallel programming should be judged by the standards of performance (serial) programming, which is also considerably more demanding than just standard serial programming. My answer is that getting good (though not the best) serial performance, as students enrolled in CS courses often manage to get and without the need to retune their code for new machine generations, should be the proper effort standard for getting significant (though not the best) parallel speedups. In other words:

- Performance in serial programming often comes through the optimizations made available by compilers, freeing the programmer from much of the burden of fine-tuning the code for performance. Such optimizations allow keeping more of the programmer's focus on getting performance by bettering the algorithm, which is not the case in parallel programming.
- The programmer can only do so much when the hardware stands in his or her way. Current computer architectures are geared toward multiple streams of serial codes (threads). For performance, a programmer is required to come up with threads that are large enough, which is something not readily available in irregular programs.

Still, this reviewer's comment and my answer suggest this Viewpoint must also demonstrate I am not dreaming, and a multicore system allowing good speedups on irregular problems and algorithms with limited effort is indeed feasible. For this reason the current version cites the XMT[a] many-core computer platform described in Vishkin,[7] which provides a useful demonstration. For example:

The world has yet to see a commercial parallel machine that can handle general-purpose programming and applications effectively.

- Students in the graduate parallel algorithms theory class I teach at the University of Maryland are required to complete five or six nontrivial parallel programming assignments, and nearly all manage to get significant speedups over their best serial version, in every assignment.
- Nearly 300 high school students, mostly from the Thomas Jefferson High School for Science and Technology in Alexandria, VA, have already programmed XMT achieving significant speedups.
- The XMT home page[b] also cites success on par, or nearly on par, with serial computing with students in middle school, an inner-city high school, and college freshmen and other undergraduate students.
- As the max-flow problem was mentioned, the XMT home page also cites a publication demonstrating speedups of over 100X over the best serial algorithm counting cycles, while speedups on any commercial system do not exceed 2.5X.
- Publications demonstrating similar XMT speedups for some of the other advanced parallel algorithms in the literature are also cited.
- XMT also demonstrates there is no conflict with backward compatibility on serial code.

The world has yet to see a commercial parallel machine that can handle general-purpose programming and applications effectively. It is up to the research community to vertically develop an integrated computing stack, and prototype and validate it with significant applications and easier parallel programming, trailblazing the way for vendors to follow. Due to its high level of risk, prototype development fits best within the research community. On the other hand, exploiting parallelism for today's commercial systems is of direct interest to industry. If indeed application development for current commercial systems will be funded by industry, more of today's scarce research funding could shift toward prototype development where it is needed most.

Ludwik Fleck, a Polish-Israeli founder of the field of sociology of science, observed the discourse of research communities is not without problems, pointing out that even the most basic consensus of such a community (for example, what constitutes a fact) merits questioning.[4] In particular, through feedback external to the community reaching consensus. This Viewpoint seeks to provide such feedback for general-purpose multicore parallelism. Its ideal impact would be driving the field toward seeking enablement of systemic advancement that will get the field out of its "rabbit hole" of limited-potential well-worn paths and ad hoc solutions, as significant and successful as these paths and solutions have been. C

a XMT stands for explicit multithreading and should not be confused with the generation of the Tera computer project, which is called Cray XMT.

b The XMT home page is http://www.umiacs.umd.edu/~vishkin/XMT/.

References

1. Ballard, G. et al. Communication efficient Gaussian elimination with partial pivoting using a shape morphing data layout. In *Proceedings of the 25th ACM Symposium on Parallelism in Algorithms and Architectures* (SPAA), (Montreal, Canada, 2013), 232–240.
2. Borkar, S.Y. et al. Platform 2015: Intel processor and platform evolution for the next decade. White Paper, Intel Corporation, 2005.
3. Edwards, J.A. and Vishkin, U. Parallel algorithms for Burrows-Wheeler compression and decompression. *Theoretical Computer Science*, to appear in 2014; see http://dx.doi.org/10.1016/j.tcs.2013.10.009.
4. Fleck, L. *The Genesis and Development of a Scientific Fact*, (edited by T.J. Trenn and R.K. Merton, foreword by Thomas Kuhn). University of Chicago Press, 1979. English translation of *Entstehung und Entwicklung einer wissenschaftlichen Tatsache. Einführung in die Lehre vom Denkstil und Denkkollektiv* Schwabe und Co., Verlagsbuchhandlung, Basel, 1935.
5. Fuller, S.H. and Millet, L.I., Eds. *The Future of Computing Performance: Game Over or Next Level.* National Research Council of the National Academies, The National Academies Press, 2011.
6. Valiant, L.G. A bridging model for parallel computation. *Commun. ACM 33*, 8 (Aug. 1990), 103–111.
7. Vishkin, U. Using simple abstraction to reinvent computing for parallelism. *Commun. ACM 54*, 1 (Jan. 2011), 75–85.

Uzi Vishkin (vishkin@umd.edu) is a professor in the Department of Electrical and Computer Engineering at The University of Maryland, College Park, and at the University of Maryland Institute for Advanced Computer Studies.

Partially supported by awards CNS-1161857 CCF-0811504 from the National Science Foundation.