SPAA'21 Panel Paper: Architecture-Friendly Algorithms versus Algorithm-Friendly Architectures

Guy Blelloch
Computer Science
Carnegie Mellon University
Pittsburgh, PA
guyb@cs.cmu.edu

William Dally
NVIDIA
Santa Clara, CA
dally@stanford.edu

Margaret Martonosi
Computer Science
Princeton University
Princeton, NJ
mrm@princeton.edu

Uzi Vishkin
University of Maryland Institute for
Advanced Computer Studies (UMIACS)
College Park, MD
Contact author: vishkin@umd.edu

Katherine Yelick
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA
yelick@berkeley.edu

ABSTRACT

The current paper provides preliminary statements of the panelists ahead of a panel discussion at the ACM SPAA 2021 conference on the topic: algorithm-friendly architecture versus architecture-friendly algorithms.

CCS CONCEPTS

- Architectures
- Design and analysis of algorithms
- Models of computation

KEYWORDS

Parallel algorithms; parallel architectures

ACM Reference format:

Guy Blelloch, William Dally, Margaret Martonosi, Uzi Vishkin and Katherine Yelick. 2021. SPAA'21 Panel Paper: Architecture-friendly algorithms versus algorithm-friendly architectures. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '21), July 6–8, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3409964.3461780

1 Introduction, by Uzi Vishkin

The panelists are:

- *Guy Blelloch, Carnegie Mellon University*
- *Bill Dally, NVIDIA*
- *Margaret Martonosi, Princeton University*
- *Uzi Vishkin, University of Maryland*
- *Kathy Yelick, University of California, Berkeley*

As Chair of the ACM SPAA Steering Committee and organizer of this panel discussion, I would like to welcome our diverse group of distinguished panelists. We are fortunate to have you.

We were looking for a panel discussion of a central topic to SPAA that will benefit from the diversity of our distinguished panelists. We felt that the authentic diversity that each of the panelist brings is best reflected in preliminary statements, expressed prior to start sharing our thoughts, and, therefore, merits the separate archival space that follows.

2 Preliminary statement, by Guy Blelloch

The Random Access Machine (RAM) model has served the computing community amazingly well as a bridge from algorithms, through programming languages to machines. This in large part can be attributed to the superhuman achievements by computer architects in maintaining the RAM abstraction as technology has advanced. These efforts include a variety of approaches for maintaining the appearance of random access memory, including sophisticated caching and paging mechanisms, as well as many developments to extract parallelism from sequences of RAM instructions, including pipelining, register renaming, branch prediction, prefetching, out-of-order execution, etc.

Based on this effort the RAM instructions used in abstract algorithms closely match actual machine instructions and language structures. It is easy to understand, for example, how the algorithmic concept of summing the elements of a sequence can be converted to a for loop at the language level, and a sequence of RAM instructions roughly consisting of a load, add to a register, increment a register, compare, and conditional jump. The RAM abstraction has greatly simplified developing efficient algorithms and code for sequential machines, but more importantly has allowed us to educate innumerable students in the art of algorithm design, programming language design, programming, application development. These students are now responsible for where we have reached in the field.

When analyzing algorithms in the abstract RAM, in all honesty, it does not tell us much about actual runtime on a particular machine. But that is not its goal, and instead the goal of such a model is to lead algorithm and application developers in the right direction with a concrete and easily understandable model that greatly narrows the search space of reasonable algorithms. The RAM by itself, for example, does not capture the locality that is needed to make effective use of caches. In many cases this locality is a second order effect---for example, when comparing an exponential vs. linear time algorithm. In some cases it is a first order effect, but it is easy to add a one level cache to the RAM model, and hundreds of algorithms have been developed in such a model. When algorithms developed in this model satisfy a property of being cache oblivious, they will also work effectively on a multilevel cache.

The RAM, however, is inherently sequential so unfortunately it fails when coming to parallelism. We need a replacement (or replacements) that supports parallelism. As with the RAM it is extremely important that this replacement(s) be simple so that it can be used to educate every computer scientist, as well as many others, about how to develop effective (parallel) algorithms and applications. As in the case of the RAM, supporting such simple models at the algorithmic and programming level will require superhuman achievements on the part of computer architects, some that have already been achieved. To be clear here, this does not mean that the instructions on the machine need to match or even have an obvious mapping from the model. There is plenty of room for languages to implement abstractions---for example, a scheduler that maps abstract tasks to actual processors. But it does mean there is some clear translation of costs from the model to the machine, even if not a perfect predictor of runtime. The goal, again, is to guide the algorithm designer and programmer in the right direction. As with extensions of the RAM, it is fine if the models have refinements that account for locality, for example, but these should be simple extensions.

At least for multicore machines, there are parallel models that are simple, use simple constructs in programming languages, and support cost mappings down to the machine level that reasonably capture real performance. This includes the fork-join work-depth (or work-span) model. There are even reasonably simple extensions that support accounting for locality, as well as asymmetry in read-write costs. It seems like the emphasis should be on supporting such simple models by adapting hardware when possible, rather than trying to push to the common user complicated and changing models capturing the technology de-jour. Without such simple models we will not be able to educate the vast majority of the next generation of students on how to effectively design algorithms and applications for future machines.

2.1 Bio

Guy Blelloch is a Professor of Computer Science at Carnegie Mellon University. He received a BA in Physics and BS in Engineering from Swarthmore College in 1983, and a PhD in Computer Science from MIT in 1988. He has been on the faculty at Carnegie Mellon since 1988, and served as Associate Dean of Undergraduate studies from 2016-2020.

His research contributions have been in the interaction of practical and theoretical considerations in parallel algorithms and programming languages. His early work on implementations and algorithmic applications of the scan (prefix sums) operation has become influential in the design of parallel algorithms for a variety of platforms. His work on the work-span (or work-depth) view for analyzing parallel algorithms has helped develop algorithms that are both theoretically and practically efficient. His work on the Nesl programming language developed the idea of program-based cost-models, and nested-parallel programming. His work on parallel garbage collection was the first to show bounds on both time and space. His work on graph-processing frameworks, such as Ligra and GraphChi and Aspen, have set a foundation for large-scale parallel graph processing. His recent work on analyzing the parallelism in incremental/iterative algorithms has opened a new view to parallel algorithms—i.e., taking sequential algorithms and understanding that they are actually parallel when applied to inputs in a random order. Blelloch is an ACM Fellow.

3 Preliminary statement, by Bill Dally. Function and space-time mapping (F&M) for harmonious algorithms and architectures

It is easy to make architectures and algorithms mutually friendly once we replace two obsolete concepts, centralized serial program execution and the RAM or PRAM model of execution cost with new models that are aligned with modern technology. With the appropriate models we can design both programmable and domain-specific architectures that are extremely efficient and easily understood targets for programmers. At the same time, we can design algorithms with predictable and high performance on these architectures.

The laws of physics dictate that modern computing engines (programmable and special purpose) are largely communication limited. Their performance and energy consumption are almost entirely determined by the movement of data. This includes memory access. Reading or writing a bit-cell is extremely fast and efficient. All the cost in accessing memory is data movement. Arithmetic and logical operations are much less expensive. In 5nm technology, an add costs about 0.5fJ/bit and a 32-bit add takes about 200ps. On-chip communication costs 80fJ/bit-mm and traveling 1mm takes about 800ps. Transporting the result of an add 1mm costs 160x as much as performing the add. Sending it across the diagonal of an 800mm2 GPU costs 4500x as much. Going off chip is an order of magnitude more expensive.

With communication paramount, efficient programs are those that optimize locality. To optimize locality requires reasoning about programs in both space and time. Each operation must be assigned a time and location, and each data element must be assigned a "path" from its definition to each use. To simplify this reasoning, it is convenient to separate the function of a computation from its mapping in time and space.

The function can be specified by a functional program that describes how each element of a computation is computed from earlier elements. No ordering - other than that imposed by data dependencies is specified. By its nature, a definition exposes all available parallelism in the computation. Any serialization - projection of a computation that could be parallel in space into one serial in time - is specified by the mapping.

The mapping specifies when and where each element is computed and where elements reside from definition to last use. The time axis can be discretized into cycles. Location can be discretized onto a grid of two or more dimensions. The delay and energy of bulk memory (DRAM, SSD, etc...) can be modeled by adding a layer to the grid.

A legal mapping is one that preserves causality - scheduling element computations after their inputs have been computed, allows time for elements to move from definition to use, and does not exceed storage bounds for elements in transit. For a dynamic computation, the mapping is determined in part at runtime. A mapping may compute the same element at multiple points in time and/or space - rather than storing it or communicating it between those points.

For a given problem - there may be several functions that compute the result (e.g., decimation in time vs decimation in space FFT, or different radix FFT). For each function there are many possible mappings that range from completely serial to minimum-depth parallel with many points between. One can systematically search the space of possible mappings to optimize a given figure of merit: execution time, energy per op, memory footprint, or some combination.

The F&M model supports modular program composition, but with constraints on mappings of input and output data structures. Functions compose as usual. Mappings, however, must be aligned to compose modules. The output of module A must have the same mapping as the input of module B for the two to be composed in series, or a remapping module must be inserted between the two to shuffle the data. Common idioms such as map, reduce, gather, scatter, and shuffle can be used by many programs to realize common communication patterns. Programmers that don't want to bother with mapping can use a default mapper – with results no worse than with today's abstractions.

This model makes it possible to write algorithms (function + mapping) with predictable execution time and energy because communication - the major source of delay and energy consumption - is made explicit, to the granularity of the grid (sub-mm).

An algorithm expressed in this model also directly specifies a domain-specific architecture. Given a definition and mapping, lowering the specification to hardware (e.g., in Verilog or Chisel) is a mechanical process.

A programmable target can be realized by putting a programmable processor at each grid point and surrounding it with many "tiles" of memory. The processor could be a serial core, a vector processor, or a CGRA. The amount of memory per processor is also a parameter that can be adjusted to tailor the architecture to a family of applications.

```
Forall i, j in (0:N-1, 0:N-1)
    H(i,j) = min(H(i-1, j-1) + f(R[i],Q[j]), H(i-1,j)+D, H(i,j-1)+ I, 0) ;
Map H(i,j) at i % P  time floor(i/P)*N + j
```

As an example, the code fragment above specifies a dynamic programming computation to compute the minimum edit distance between two strings R and Q of length N. The function is just the recurrence equation for H(i,j). The mapping places this on array of P processors as marching anti-diagonals.

The function and mapping (F&M) model makes algorithms and architectures work well together because it embraces the laws of physics - making the parallelism in the algorithm and the cost of communication explicit. In contrast, conventional serial architectures, and algorithms developed assuming the RAM or PRAM model ignore the laws of physics - hiding communication, and parallelism - with very high cost and difficult to predict performance.

A modern multicore CPU hides the two physical realities of parallelism and spatially distributed memory. Each core is a parallel engine – issuing up to 8 instructions per cycle and having hundreds of instructions (size of ROB) in flight at a time. The cost of this is a 10,000x loss of efficiency. The energy overhead of an ADD instruction is 10,000x times more than the energy required to do the add. Similarly, the spatial nature of memory is hidden. Adding two numbers that are co-located at a distant point requires first transporting them to the processor – again at a cost of 1,000x or more the energy of doing the addition at the remote point.

The RAM and PRAM models that are used to analyze and compare algorithms hide the reality of spatial distribution and the huge difference between computing and communication costs. In these models, everything is unit cost. An add operation costs the same as transporting data from off-chip memory – even though the off-chip access is 50,000x more expensive. Yes, this is a constant factor, but constant factors are important. When comparing two FFT algorithms that are both O(NlogN), the one that is 50,000x more efficient is preferred.

Much work has addressed communication costs: Demmel's communication avoiding algorithms, cache-oblivious algorithms, weight-stationary dataflows for DNN accelerators, systolic arrays, among others. However, none of these have addressed the problem in a general manner.

Almost all demanding problems admit solutions that are highly parallel and that minimize communication. Our current abstraction of serial processors and PRAM models, however, hide parallelism and communication making it difficult to optimize these critical aspects of the program. The F&M model separates a program's function from its mapping, exposes all parallelism, and makes communication explicit. This makes it easy to predict the cost and performance of a program – and hence to optimize it. Such programs can be mapped to accelerators that are >10,000x or more efficient than conventional architectures. Alternatively, they can be targeted to programmable architectures that are 100s of times more efficient.

The F&M model raises a host of research questions across computer science. What languages best express functions and mapping and facilitate abstraction and modular composition of programs? What methods can best find optimal mappings? What programmable architectures are good targets for these programs? Perhaps most importantly, what are the best algorithms given this realistic model of costs?

3.1 Bio

Bill is Chief Scientist and Senior Vice President of Research at NVIDIA Corporation and an Adjunct Professor and former chair of Computer Science at Stanford University. Bill is currently working on developing hardware and software to accelerate demanding applications including machine learning, bioinformatics, and logical inference. He has a history of designing innovative and efficient experimental computing systems. While at Bell Labs Bill contributed to the BELLMAC32 microprocessor and designed the MARS hardware accelerator. At Caltech he designed the MOSSIM Simulation Engine and the Torus Routing Chip which pioneered wormhole routing and virtual-channel flow control. At the Massachusetts Institute of Technology his group built the J-Machine and the M-Machine, experimental parallel computer systems that pioneered the separation of mechanisms from programming models and demonstrated very low overhead synchronization and communication mechanisms. At Stanford University his group developed the Imagine processor, which introduced the concepts of stream processing and partitioned register organizations, the Merrimac supercomputer, which led to GPU computing, and the ELM low-power processor. Bill is a Member of the National Academy of Engineering, a Fellow of the IEEE, a Fellow of the ACM, and a Fellow of the American Academy of Arts and Sciences. He has received the ACM Eckert-Mauchly Award, the IEEE Seymour Cray Award, the ACM Maurice Wilkes award, the IEEE-CS Charles Babbage Award, the IPSJ FUNAI Achievement Award, the Caltech Distinguished Alumni Award, and the Stanford Tau-Beta-Pi Teaching Award. He currently leads projects on computer architecture, network architecture, circuit design, and programming systems. He has published over 250 papers in these areas, holds over 160 issued patents, and is an author of the textbooks, *Digital Design: A Systems Approach*, *Digital Systems Engineering*, and *Principles and Practices of Interconnection Networks.*

4 Preliminary statement, by Margaret Martonosi. Seismic shifts: Challenges and opportunities in the "post-ISA" era of computer systems design

For decades, Moore's Law and its partner Dennard Scaling have together enabled exponential computer systems performance improvements at manageable power dissipation. With the recent years of slowing of Moore/Dennard improvements, designers have turned to a range of approaches for extending scaling of computer systems performance and power efficiency. These include specialized accelerators and heterogeneous parallelism. In most current processor chips, far more area is devoted to specialized accelerators than to the ``main'' CPU.

Unfortunately, the scaling gains offered by specialization and heterogeneity come with significant costs: increased hardware and software complexity, degraded programmability and portability, and increased likelihood of design errors and security vulnerabilities. In the process of advancing hardware generation-by-generation, our systems design approaches are moving away from abstraction. The long-held hardware-software abstraction offered by the Instruction Set Architecture (ISA) interface is fading quickly in this post-ISA era.

In this panel, I will talk about the new opportunities that emerge in this post-ISA era. Namely, in navigating today's specialization trends, we have the opportunity to define new interfaces and consider new techniques for mapping from algorithm to architecture. Drawing from recent research examples, I will advocate for a shift towards formal specifications that support automated full-stack verification for correctness and security.

4.1 Bio

Margaret Martonosi is the Hugh Trumbull Adams '35 Professor of Computer Science at Princeton University, where she has been on the faculty since 1994. Martonosi's work has included the widely-used Wattch power modeling tool and the Princeton ZebraNet mobile sensor network project for the design and real-world deployment of zebra tracking collars in Kenya. Her current research focuses on computer architecture and hardware-software interface issues in both classical and quantum computing systems. Martonosi is a member of the National Academy of Engineering and the American Academy of Arts and Sciences, in addition to being a Fellow of IEEE and ACM.

5 Preliminary statement, by Uzi Vishkin. Give me a killer-app and you will get PRAM-supporting general-purpose many-core

Give me the place to stand, and I shall move the earth, Archimedes

One may wonder: why has the panel topic even become an issue? After all, the serial random-access-machine (RAM) algorithmic model has guided the specs for serial CPUs since their advent, making them algorithm-friendly by design. Perhaps due to the von-Neumann architecture, this created a clear distinction: algorithm and application developers were "customers" whose requirements computer system developers needed to meet.

Parallel algorithms and parallel programming tend to be more demanding than their serial counterparts. So, it would only be natural to adopt similar standards, if not higher, for parallel computer systems. However, recent calls for architecture-friendly algorithms (e.g., [1]) turn this on its head. Not only that programming parallel architectures, and now accelerators, is more involved, this idea enforces architectures and accelerators as the customers on algorithm and application developers. Would it work for vendors to tell their customers that roles have been reversed? I also believe that raising the skill bar for direct effective programming of commodity platforms to what appears to be PhD-level is a retreat for the field. For example, consider market and social objectives such as increased diversity of applications and developers. Jensen Huang, CEO, NVIDIA, has argued not far from the opposite: due to AI we are in a new age of computing where SW writes SW, since it is too difficult for humans to do. But, isn't the ability of AI to produce SW limited, at least for the foreseeable future? Yet, with few exceptions CS undergraduate students are not trained to program for performance the parallelism on their own laptop or desktops. It is also too difficult for most CS professionals to exploit such parallelism. Continued need for many-core-friendly algorithms will likely continue to repel algorithm practitioners [7], though academics may see hard to program platforms as publication opportunities.

The decline of computers as general purpose technology was recently reviewed in [5]. Previously, [2] stated: "at the present, it seems unlikely that form of simple multicore scaling will provide a cost-effective path to growing performance". I will argue later why these two references provide critical context for raising a worrisome possibility: a monopoly in the many-core space by a platform that is not algorithm friendly.

However, first, I point out that simple general-purpose many-core scaling is feasible. The extensive FPGA-based prototyping of the XMT PRAM-on-chip platform at UMD [6], culminating in advanced compiler [4], have shown feasibility of a competitive scalable general-purpose many-core, in comparison, e.g., with GPUs, for as-is complete PRAM algorithms for the same problem. Balancing speed, power, silicon area and locality was key to establishing competitive advantage over hardware-centered approaches. So, while I believe that our UMD team work established the utility of especially irregular PRAM algorithms and the cost-effectiveness of the PRAM abstraction, the comment on lack of a cost-effective path still has merit. Even if the feasibility of a cost-effective system target can be established, a cost-effective path may not be that obvious, as another obstacle stands in the way: the expectation of timely emergence of complete applications that would merit investment in the system. Namely, the ease of programming for a system supporting PRAM algorithms in itself does not provide sufficient promise to investors for timely emergence of such applications. It appears that the magnitude of this obstacle has escaped attention of most computer scientists, whether practitioners or theorists.

Second, GPUs form the most commercially successful form factor for many-core processors to date. Their original computer graphics application allowed GPUs to first overcome this meritorious application obstacle. Once available, it was possible to expand their footprint by developing more applications. The serendipitous GPU support for deep learning stands out as an example. Would the connection between GPUs and deep learning applications have been discovered (and developed) had GPUs not been already available? I doubt it. But, once the bridgehead of GPUs for AI has been established, later AI-targeted software systems could expand them for more applications.

Third, many-core computing can offer improvement by 4-5 orders of magnitude over single cores; but: does a **chicken-and-egg impasse** stand in the way of realizing non-GPU many-core platforms and their applications? More specifically, are there potential applications that could have benefited from other forms of many-core computing that have not been developed? The likely reason being that investment in such applications prior to having a sufficiently mature many-core platform is as challenging as the development of such a platform prior to having sufficiently mature applications for them. At present GPUs are the only form factor whose investment worth can be supported by sufficient money-making applications. Thus, this chicken-and-egg impasse brings to the fore a critical question for any many-core different than GPUs, whether general-purpose or an accelerator. Hence the monopoly concern.

My forth point is that application development research can potentially shape commodity computing platforms of the future. While the creation of new prototype "killer apps" is challenging, it can advance the whole chain of many-core innovation. *The possibility of some new killer app(s) that would drive a general purpose many-core platform sounds to me most intriguing*, as it could reverse the general purpose decline trajectory noted above. If achieved, the impact will be felt across the information technology sector and beyond, giving the national economy that will lead it a strategic advantage over competing economies for decades to come. But to achieve that government funding needs to recognize the merit of such a quest for killer app development. Per [5] applications that are left behind by today's specialized processors include those that: (i) get little performance benefit, (ii) are not a sufficiently large market to justify upfront fixed costs, or (iii) cannot coordinate their demands; this last category is where general purpose platforms appear to belong: *enough applications need to be aligned for justifying such upfront cost.* If done, algorithm-friendly architectures could offer a more cost-effective, general purpose PRAM-based path, thereby a quantum leap in application innovation.

Fifth, another important objective should be: *whatever will drive the more apps.* It will be up to bespoke architectures as well as general purpose ones to compete in this category.

5.1 Bio

In lieu of a generic bio, the remainder of this text will review some of the evolution of the field through Uzi Vishkin's personal prism, explaining what drove him to developing a reconciliation approach to algorithm-friendly and architecture-friendly aspirations, and therefore written in first person. Circa 1970 David Kuck advocated automatic parallelization of serial code using compilers only. As I first considered parallel computing as my future field of research in late 1979, I expected that this compiler line of work would make significant headway on regular programs, where memory access is largely data independent, but I was skeptical regarding its potential to affect the type of data structures and algorithms that formed the core curriculum of CS education. I felt that parallel algorithms would form an alien culture to serial algorithms, and doubted the ability of automatic parallelization efforts by a compiler to disambiguate parallelism inherent in serial algorithms. For example, breadth-first search on graphs had been tied to a first-in first-out queue for no good reason other than enforcing serialization, even where parallelism exists, in part because such parallelism would imply limited non-determinism. I also felt that theoretical computer science was not ready. Still excited by the profound insights of the theory of NP-completeness, the primary objective in the mind of theorists at the time was developing a related complexity theory for parallelism around P-completeness. Their focus was on poly logarithmic time algorithms using a polynomial number of processors, known as the NC theory, even when a linear time serial algorithm was known for a problem. However, providing speedups over serial processing has been the main reason for parallel computing. I did not see how a polynomial number of processors, say n^3 processors, could realistically offer speedups.

I was privileged to have Nobel Laureate R.J. Aumann, as my MS thesis advisor, teaching me to figure out which problems are fundamental *on my own.* I recall well how in 1979 these compiler and complexity backdrops did not prevent me from betting my career on an independent direction: work efficient PRAM algorithms. In retrospect, this worked out fine: (i) In 1984 this direction was recognized as mainstream theory, almost overnight; and (ii) By the end of the 1980s, several surveys were already able to summarize a considerable volume of parallel algorithms work with more fundamental work done into the early 1990s. In 1996, my ACM Fellow citation recognized my contributions. For me, this independence continued with the XMT/PRAM-on-Chip platform that I have led at UMD since the late 1990s. The underlying premise has been that the only domain in which a programmer can outdo a compiler is by expressing the work-depth-type parallelism that the underlying abstract algorithm provides. On nearly all other issues (e.g., locality), compiler transformations will (eventually) have the upper hand. I recall a conversation with compiler experts who inspired me with the following message: "you studied math, so as long as you can provide a non-ambiguous optimization objective that looks to you feasible, we, compiler guys, will likely be able to deliver it". Having invented the XMT architecture [6], which to a first approximation is about reducing overheads of PRAM algorithms using hardware primitives, I was fortunate to have quite a few talented students develop simulators, committing the architecture to silicon (FPGA) and studying its interconnection networks and power, as well as applications. They all deserve recognition for their contributions. However, most relevant to the current statement is the chain of compiler work. (i) My parallel algorithms and programming students had sweat hard to tune abstract parallel algorithms into becoming architecture-friendly. (ii) Looking over their shoulders to gain insights, several generations of compiler students (mostly co-advised by compiler expert Rajeev Barua) incorporated these tunings into (or "taught them to") the compiler. Indeed, the performance of the most recent architecture-friendly compiled code [4] has been as good as the best manually optimized code produced by expert programmers.

Reflecting on the title for this panel, I believe that my team's work demonstrated how to reconcile architecture-friendly algorithms and algorithms-friendly architectures as follows. Use *architecture-friendly compilers* to enable algorithms-friendly systems.

6 Preliminary statement, by Katherine Yelick. Algorithmic players in the Moore's Law end game

Single processor clock speed scaling ended over a decade ago and transistor sizes are approaching atomic scaling limits, while the demand for computing in science, engineering, business, and government continues to grow. *Parallelism* will be the main source of performance increase, from scaling out larger clusters, increasing core counts from simpler cores and larger chips, to wider data parallelism within cores. There are several reasons to increase parallelism, from devices to programming models. At the device level, lower energy designs may lead to higher latencies and thus require more parallelism to increase overall performance. Latency of data movement costs between compute nodes and within local memory systems have been relatively stagnant and thus contribute to another demand for increased parallelism to hide latencies. A desire for portability and ease of programming leads to virtualized models, often expressing more parallelism than is required for a particular hardware configuration, so one does not have to program to a fixed number of cores, threads, cache sizes, or communication buffer sizes.

These fundamental design constraints will require algorithms with more parallelism at all levels but also *simpler* and more efficient software and hardware mechanisms to create, synchronize and manage that parallelism. Just as smaller, simpler cores have led to better overall performance when paired with scalable parallel algorithms, we need simpler mechanisms for communication and synchronization, avoiding unnecessary memory copying, ordering constraints, and blocking of useful work. Heavyweight communication mechanisms that imply global or pairwise synchronization and require more data aggregation to amortize overhead can consume precious fast memory resources. Today's machines have more levels of memory and more domains of communication, more required software control, and more architectural diversity in the underlying synchronization and communication mechanisms, which are significant challenges to portability. Algorithm designers could have significant influence in showing that a simpler set of data movement and synchronization primitives are universally useful across algorithms and applications. While this exists to some extent for shared memory systems, they are not as clear or well accepted for distributed memory systems or heterogeneous node architectures.

There is a significant gap between communication and computation cost in both running time and energy use. This has grown over decades while compute performance improved whether through clock speed or parallelism, and off-chip latency and bandwidth improvements did not keep pace. Algorithms must also treat *communication avoidance* as a first-class optimization target, reducing both data movement volume and number of distinct events, while being cognizant of consuming memory resources.

A final trend in hardware—which desperately needs the input of algorithm and application developers—is *specialization.* One can better take advantage of parallelism and simplicity if designing hardware for a particular algorithmic target. Computers designed of genomics may not need floating point but require fine-grained communication, some machine learning algorithm may take advantage of lower precision floating point, or at the extreme, problems in chemistry or physics may take advantage of quantum devices. We are seeing specialization today in the wide-spread interest in processors for deep learning, often designed specifically for applications like image analysis using convolutional neural networks (CNNs). There is a danger that, just as the Linpack benchmark may have had disproportionate influence on parallel computers designed for modeling and simulation, this single-minded focus on a particular class of algorithms may both limit hardware innovations and ultimately constrain the class of algorithms and applications that benefit from computing performance growth. From a practical standpoint, the enormous power of market pressures for hardware with significant commercial interests cannot be ignored. But algorithms and software developers as well as architects should continue to explore the breadth of application drivers that are important for societal, human, and scientific domains, looking for overlap in computational motifs and ensuring a robust, diverse and healthy future landscape for parallel hardware, software and algorithms.

6.1 Bio

Katherine Yelick is the Robert S. Pepper Distinguished Professor of Electrical Engineering and Computer Sciences at UC Berkeley, where she is also the Associate Dean for Research in the Division of Computing, Data Science, and Society. She earned her Ph.D. from MIT and has been on the faculty at UC Berkeley since 1991. Yelick was the Director of the NERSC supercomputing facility from 2008 to 2012 and the Associate Laboratory Director for Computing Sciences at Lawrence Berkeley National Laboratory from 2010 through 2019. She is an ACM Fellow, AAAS Fellow, and recipient of the ACM-W Athena and ACM/IEEE Ken Kennedy awards. She is a member of American Academy of Arts and Sciences and the National Academy of Engineering.

REFERENCES

[1] W.J. Dally, Y. Turakhia, S. Han. Domain-Specific Hardware Accelerators. CACM 63,7 48-57, 2020. "We envision future programming systems where the programmer specifies the algorithm and a mapping to hardware in space and time."

[2] J.L. Hennessy and D.A. Patterson. Computer Architecture A Quantitative Approach, 6th Edition. Morgan Kaufmann, 2017.

[3] J.L. Hennessy and D.A. Patterson. A new golden age for computer architecture. CACM 62,2 48-60, 2019. "The next decade will see a Cambrian explosion of novel computer architectures…"

[4] F. Ghanim, R. Barua and U. Vishkin. Easy PRAM-based High-performance Parallel Programming with ICE. *IEEE Transactions on Parallel and Distributed Systems 29:2, 2018.*

[5] N.C. Thompson and S. Spanuth. The Decline of Computers as a General Purpose Technology. CACM 64,3 64-72, 2021.

[6] U. Vishkin. Using Simple Abstraction to Reinvent Computing for Parallelism. CACM, 54,1 75-85. 2011.

[7] U. Vishkin. Is Multicore Hardware for General-Purpose Parallel Processing Broken? *CACM* 57,4 35-39, 2014.