DOI:10.1145/3554977

TURING LECTURE

Jack. J. Dongarra is the recipient of the 2021 ACM A.M. Turing Award, recognized for his pioneering contributions to numerical algorithms and libraries that enabled high-performance computational software to keep pace with exponential hardware improvements for over 40 years.

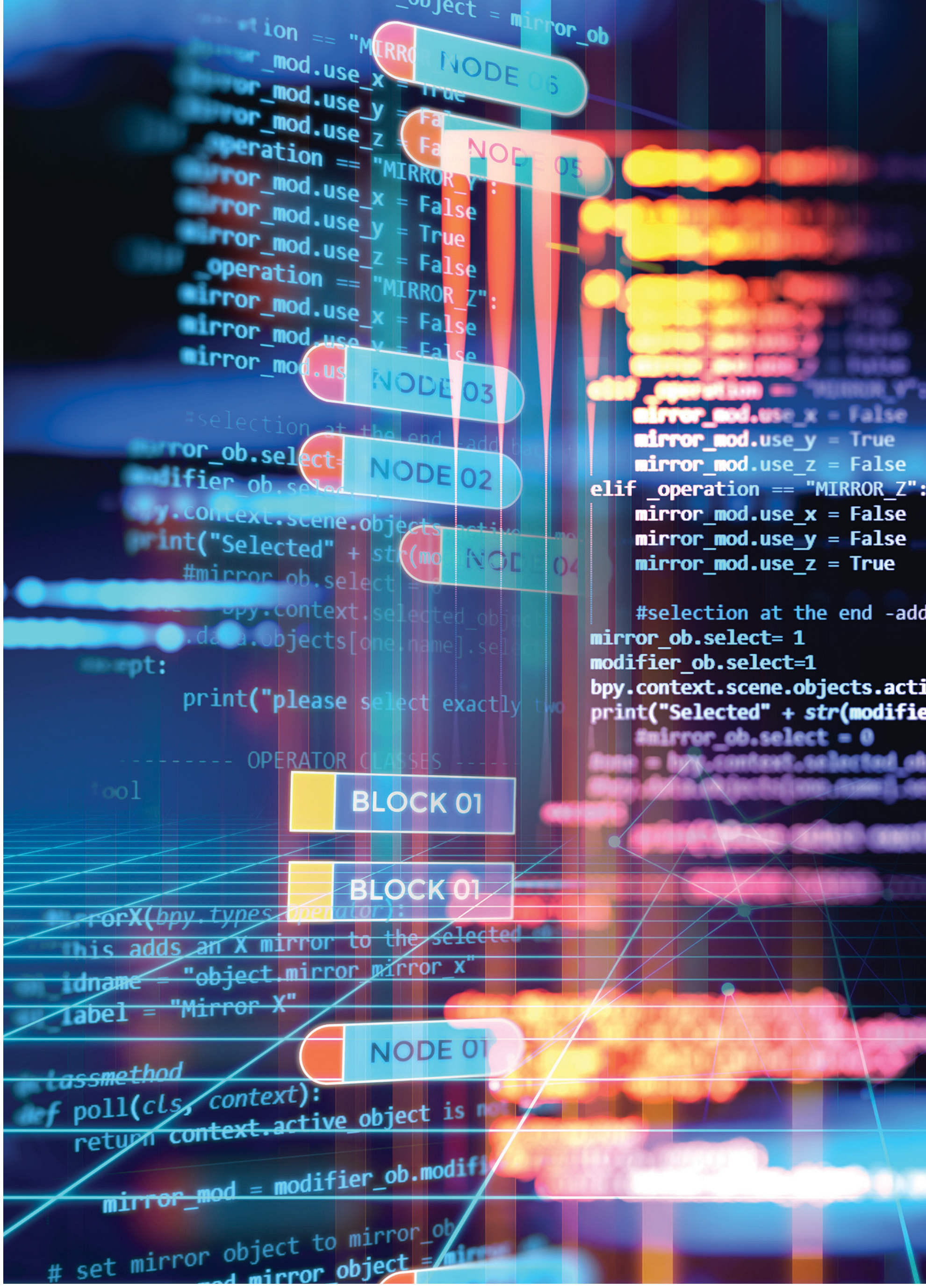
Tracing how software and algorithms follow the hardware.

BY JACK J. DONGARRA

The Evolution of Mathematical Software

OVER FOUR DECADES have passed since the concept of computational modeling and simulation as a new branch of scientific methodology—to be used alongside theory and experimentation—was first introduced. In that time, computational modeling and simulation has embodied the enthusiasm and sense of importance that people in our community feel for the work they are doing. Yet, when we try to assess how much progress we have made and where things stand along the developmental path for this new "third pillar of science," recalling some history about the development of the other pillars can help keep things in perspective. For example, we can trace the systematic use of experiment back to Galileo in the early 17th century. Yet for all the incredible successes it enjoyed over its first three centuries, the experimental method arguably did not fully mature until the elements of good experimental design and practice were finally analyzed and described in detail in the first half of the 20th century. In that light, it seems clear that while computational science has had many remarkable successes, it is still at a very early stage in its growth.

Many of those who want to hasten that growth believe the most progressive steps in that direction require much more community focus on the vital core of computational science: software and the mathematical models and algorithms it encodes. Of course, the general and widespread obsession with hardware is understandable, especially given exponential increases in processor performance, the constant evolution of processor architectures and supercomputer designs, and the natural fascination that people have for big, fast ma-



chines. I am not exactly immune to it. When it comes to championing computational modeling and simulation as a new part of the scientific method, the complex software "ecosystem" that coincides must be forefront.

At the application level the science must be captured in mathematical models, which in turn are expressed algorithmically and ultimately encoded as software. Accordingly, on typical projects much of the funding goes to support this translation process that starts with scientific ideas and ends with executable software, and which over its course requires intimate collaboration among domain scientists, computer scientists, and applied mathematicians. This process also relies on a large infrastructure of mathematical libraries, protocols, and system software that has taken years to build up and that must be maintained, ported, and enhanced for many years to come if the value of the application codes that depend on it are to be preserved and extended. The software that encapsulates all this time, energy, and thought, routinely outlasts (usually by years, sometimes by decades) the hardware it was originally designed to run on.

Thus, the life of computational science revolves around a multifaceted software ecosystem. Domain scientists now want to create much larger, multidimensional applications in which a variety of previously independent models are coupled together, or even fully integrated. They hope to be able to run these applications on exascale systems with tens of thousands of processors, to extract all performance that these platforms can deliver, and to do all this without sacrificing good numerical behavior or programmability.

High-performance computers continue to increase in speed and capacity, with exascale machines here in 2022.[30] Alongside these developments, architectures are becoming progressively more complex, with multi-socket, multi-core central processing units (CPUs), multiple graphics processing unit (GPU) accelerators, and multiple network interfaces per node. This new complexity leaves existing software unable to make efficient use of the increased processing power.

For decades, processor performance has been improving in each generation consistent with Moore's Law doubling transistor counts every two years and Dennard Scaling[6] enabling increases in clock frequency. Combined, these doubled peak performance every 18 months. Since Dennard Scaling ceased around 2006 due to physical limits, the push has been toward multi-core architectures. Instead of getting improved performance for "free" through hardware improvements, software had to be adapted to parallel, multi-threaded architectures.

In addition to multi-threaded CPU architectures, hybrid computing has also become a popular approach to increasing parallelism, with the introduction of CUDA in 2007 and OpenCL in 2009. Hybrid computing couples' heavyweight CPU cores (using out-of-order execution, branch prediction, hardware prefetching, among others) with comparatively lighter weight (using in-order execution) but heavily vectorized GPU accelerator cores. There is also heterogeneity in memory: large, relatively slow CPU DDR memory coupled with smaller but faster GPU memory such as 3-D stacked high-bandwidth memory (HBM). To take advantage of these capabilities, modern software must explicitly program for multicore CPUs and GPU

IMAGE BY WHITEMOCCA

accelerators while also managing data movement between CPU and GPU memories and across the network to multiple nodes.

The compute speed, memory and network bandwidth, and memory and network latency increase at different exponential rates, leading to an increasing gap between data movement speeds and computation speeds. For decades, the machine balance of compute speed to memory bandwidth has increased 15%–30% per year (see the accompanying figure). Hiding communication costs is thus becoming increasingly more difficult. Instead of just relying on hardware caches, new algorithms must be designed to minimize and hide communication, sometimes at the expense of duplicating memory and computation.

Very high levels of parallelism also mean that synchronization becomes increasingly expensive. With processors at around 1GHz–3GHz, exascale machines, with 1018 64-bit floating point operations per second, must have billion-way parallelism. This is currently anticipated to be achieved by roughly 2GHz × 10,000 nodes × 100,000 thread-level and vector-level parallelism. Thus, the computational and communication parallelism must become asynchronous and dynamically scheduled.

Mathematical libraries are, historically, among the first software adapted to the hardware changes occurring over time, both because these low-level workhorses are critical to the accuracy and performance of many different types of applications, and because they have proved to be outstanding vehicles for finding and implementing solutions to the problems that novel architectures pose. We have seen architectures change from scalar to vector to symmetric multiprocessing to distributed parallel to heterogeneous hybrid designs over the last 40 years. Each of these changes has forced the underlying implementations of the mathematical libraries to change. Vector computers used Level 1 and Level 2 basic linear algebra subprograms (BLAS);[4] with the change to cache-based memory hierarchies, algorithms were reformulated with block operations using Level 3 BLAS matrix multiply. Task-based scheduling has addressed multicore CPUs, while more recently—as the compute-speed-to-bandwidth ratio increases—algorithms have again been reformulated as communication avoiding. In all these cases, ideas that were first expressed in research papers were subsequently implemented in open source software, to be integrated into scientific and engineering applications, both open source and commercial.

Developing numerical libraries that enable a broad spectrum of applications to exploit the power of next-generation hardware platforms is a mission-critical challenge for scientific computing generally, and for HPC specifically. This challenge raises a variety of difficult issues. For instance, programming models and hardware architectures are still in a state of flux, and this uncertainty is bound to inhibit the development of libraries as new configurations and abstractions are tried. It is additionally prudent to expand on existing libraries instead of developing entirely new ones, when possible, as this will disperse some of the software maintenance costs, provide backward compatibility, and make transition for applications easier. Introducing radically different algorithms and methods at low levels, without radically altering usage characteristics of familiar packages at high levels, remains a software engineering conundrum. Moreover, many HPC applications will need to run on platforms ranging from leadership-class machines to smaller-scale clusters, workstations, and even laptops. These architectural changes have come every decade or so, thereby creating a need to rewrite or refactor the software for the emerging architectures. Scientific libraries have long provided a large and growing resource for high-quality, reusable software components upon which applications can be rapidly constructed—with improved robustness, portability, and sustainability.

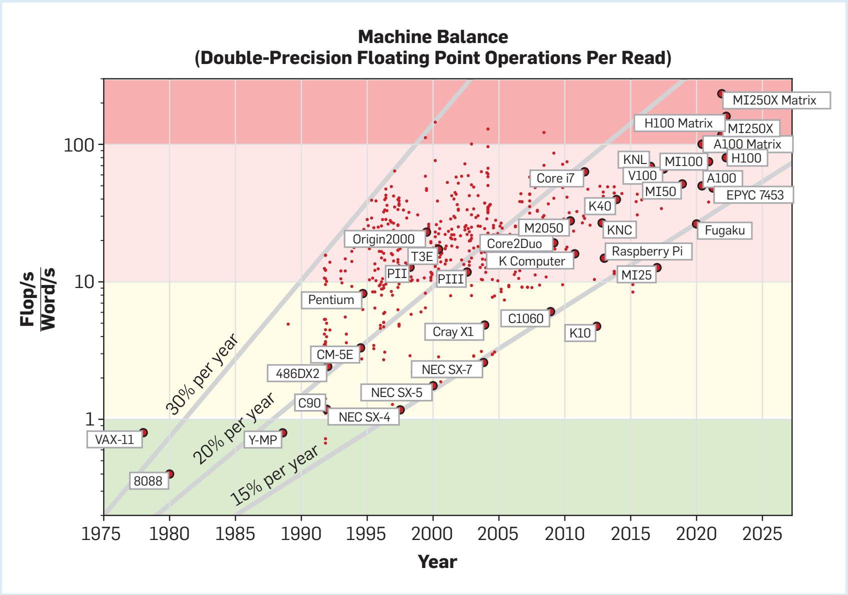
Background

Today's scientists often tackle problems that are too abstruse to parse theoretically, or too hazardous to tackle experimentally. How can a researcher peer inside a star to see exactly how it explodes? Or how can one predict impacts of climate change with so many variables?

Simulations using high-performance computers have thus become a critical resource for research in all scientific domains. But scientists first need to express their problem in a mathematical language that the computer can understand.

Standards are critical for software development. Research has always benefited from the open exchange of ideas and the opportunity to build on the achievements of others. Standards such

Processor and machine balance increasing, making communication relatively more expensive. Plot for 64-bit floating point data movement and operations; bandwidth from CPU or GPU memory to registers. Data from vendor specs and STREAM benchmark.[26]



as MPI, the BLAS, IEEE floating point standards, and numerical libraries are built on the experience of a wider community and based on best practices.

BLAS. Since the early days of HPC, the Level 1, Level 2, and Level 3 BLAS standards[13–16,25] abstracted away the low-level hardware details from scientific library developers by encoding high-level mathematical concepts like vector, matrix-vector, and matrix-matrix products.

Critical to effective high-performance computing, avoiding unnecessary memory movement has provided considerable motivation for devising algorithms that minimize data movement. Along these lines, much activity in the past 30 years has involved the redesign of basic routines in linear algebra, using block algorithms based on matrix-matrix techniques.[3] These have proved effective on a variety of modern computer architectures with vector processing or parallel-processing capabilities, on which high performance can potentially be degraded by excessive transfer of data between different levels of memory (for example, registers, cache, main memory, and solid-state disks).

By organizing the computation into blocks, we provide for full reuse of data while each block is held in cache or local memory, avoiding excessive movement of data, and giving a *surface-to-volume effect* for the ratio of data movement to arithmetic operations, that is, $O(n^2)$ data movement to $O(n^3)$ arithmetic operations. In addition, parallelism can be exploited in two ways:

- operations on distinct blocks may be performed in parallel; and
- within the operations on each block, scalar or vector operations may be performed in parallel.

More ops ≠ more time. Complexity theory clearly dictates that fewer operations, especially at a lower asymptotic bound, are preferable for optimal execution time. In high performance and scientific computing, a similar guideline was applied when every cycle and every instruction was at a premium. But this was the case in the single-core world, and it has already changed in the multicore era. Worse yet, it's further exacerbated in the case of hardware accelerators with total compute power exceeding 1,000GFLOPs in double precision and bandwidth topping at 200Gbytes/s. An order of magnitude more operations must be performed for every byte that arrives from the main memory. Computation is fast only when it happens in processor registers—even the fastest cache needs a handful of clock cycles to deliver data items.

Compared to the main memory that holds most data structures, operations on registers are virtually free, with data movement and synchronization being the essential factors contributing to algorithm speed. Projections for future machines only exacerbate the current data movement crisis. Even with the newly introduced stacked memory that promises a mind-boggling 1Tbytes/s of bandwidth, computing devices will eventually achieve performance levels more than 10TFLOPs, and the bandwidth/compute imbalance will become even more pronounced. In such an environment, we must abandon the notion that knowing the number of operations for an algorithm is a good indicator of its ultimate performance. Rather, we must look critically at the kind of operations that are required. And above all, we must focus on data movement, synchronization points, and understanding of the nature of the interaction between threads and processes in the system to make sure that they can proceed on their own for as long as possible without costly communication. In addition, we must examine the amount of data the algorithm accesses and choose one that can minimize accesses—we call this approach "communication avoiding."

Software PACKs. Delivering specialized scientific software in the form of packages, such as EISPACK,[20] LINPACK,[12] LAPACK,[3] ScaLAPACK,[9] and others, continues to be essential for delivering robust solvers that enable portable performance across ever more specialized hardware systems.

The portability of software library code has always been an important consideration, made much more difficult by diverse modern hardware designs and the corresponding flourishing of a diverse programming language landscape. Understandably, scientific teams do not wish to invest significant effort to port large-scale application codes to each new machine when the teams are focused on science results rather than software engineering. Our answer to this glaring problem has always been the development of performance-portable software libraries that hide many machine-specific details yet allow automated adaptation to the user's platform of choice.

LAPACK[3] is an example of a mathematical software package wherein the highest-level components are portable, while machine dependencies are hidden in lower-level modules. Such a hierarchical approach is probably the closest one can come to software performance-portability across diverse parallel architectures. The BLAS that LAPACK heavily relies on provide a portable, efficient, and flexible standard for application programmers.

Portable performance layers. The layered approach to performance portability is indispensable for building ever more intricate libraries on top of a less complex portability layer with desirable performance characteristics. The first mathematical subroutine library for a computer was written by Maurice V. Wilkes, David J. Wheeler, and Stanley Gill for the EDSAC at the University of Cambridge in England in 1951.[33] The programs were written in machine language, and certainly no thought was given to portability; to have a library at all was remarkable. That was followed by the foundational book edited by Wilkinson and Reinsch that described the algorithms in Algol.[34] Intuitively, our notion of portable numerical software is quite clear: portable applications successfully run on a variety of computer architectures and configurations.

Examples of different computer architectures include single processor with uniform random-access memory, pipeline or vector computers, parallel computers, and heterogeneous or hybrid computers, to name a few. Different versions of a library routine may be written for different architectures, where each version has the same calling sequence interface. Or the library routine may have the ability to determine which architecture to run on and to choose which path to take to execute on the underlying architecture successfully and efficiently. Applications use these numerical libraries, and it is these libraries we expect to be portable across different architectures.

Specific techniques and approaches. Here, we cover the transitional process for mathematical software in greater detail.[17]

Dataflow scheduling. In the late 1970s, dataflow scheduling was realized for mapping programs represented as a direct acyclic graph (DAG) of tasks to a specialized hardware configuration of systolic arrays.[24] In the ensuing decades, many task-based runtime systems have been proposed and remain active[2,5,7,8,11,23,28] with an overarching purpose to address programmability and management of parallelism in the context of HPC. The next step is to turn the dataflow scheduling approach into a standard akin to MPI.

Communication avoiding algorithms. The new normal in HPC may be summarized as follows: compute time depends on memory accesses and not on total operation count. In other words, the number of arithmetic instructions executed no longer directly reflects the time spent in running the program; the type of operation is the essential aspect to consider. Opting for higher complexity algorithms may be preferable if the operations better fit the hardware and transfer less data across the modern memory hierarchy and on-node interconnects.[21,22] To better represent the execution time of software, the performance model must be a function of both computation and communication costs. To address the computation-communication imbalance, several communication-avoiding (CA) algorithms have been developed by redesigning existing methods to obtain the minimum theoretical communication cost for a particular solver,[4,27] including CALU and CAQR factorization algorithms.[10] After basic research established their advantages, communication avoiding algorithms are now being integrated into various libraries such as LAPACK, ScaLAPACK, MAGMA, SLATE, and vendor libraries.

To better represent the execution time of software, the performance model must be a function of both computation and communication costs.

Mixed precision. The emergence of deep learning as a leading computational workload on large-scale cloud infrastructure installations has led to a plethora of heavily specialized hardware accelerators that can tackle these types of problems. These new platforms offer new 16-bit floating-point formats with reduced mantissa precision and exponent range at significantly higher throughput rates, which makes them attractive in terms of improved performance and energy consumption. Mixed-precision algorithms are being developed to leverage these significant advances in computational power, while still maintaining accuracy and stability on par with the classic single or double precision formats through careful consideration of the numerical effects of half precision. Even though research on mixed-precision algorithms has been presented in papers and conferences over the last few decades, these techniques mostly remained in a prototype state and rarely made it into production code. Recently, the U.S. Department of Energy (DOE) Exascale Computing Project (ECP) has allocated resources to bring these techniques into production.[1]

Approximate, randomized, and probabilistic approaches. In the past, the main goals for robust high-performance numerical libraries were accuracy first and efficiency second. The current outlook, informed by application needs, has been transforming rapidly: accuracy itself is often a tunable parameter. It is now one of the major contributors to excessive computation and is therefore directly at odds with speed. In a wide range of applications, from high-performance data analytics (HPDA) to machine/deep learning, and from edge sensors producing extreme amounts of data (including redundant or faulty data) to large data stores, the modern requirement for various optimizations is to establish a "best" solution in a limited period. This realignment of priority motivates the development of algorithms that call for approximations, randomization, probabilistic accuracy, and convergence bounds. The preferred algorithms compute quickly while still being sufficiently accurate through non-traditional, innovative approaches. Here we see distinct feedback from application needs back to the development of new algorithms.

Machine learning/autotuning. Although Moore's Law is still in effect, the multicore and accelerator revolution has initiated a processor design trend of moving away from architectural features that do not directly contribute to processing throughput. This means a preference toward shallow pipelines with in-order execution and cutting down on branch prediction and speculative execution. On top of that, virtually all modern architectures require some form of vectorization to achieve top performance, whether it be short-vector, single instruction, multiple data (SIMD) extensions of CPU cores or single instruction, multiple threads (SIMT) pipelines of GPU accelerators. With the landscape of future HPC populated with complex, hybrid vector architectures, automated software tuning could provide a path toward portable performance without heroic programming efforts.[32]

Impact and Lessons Learned

Measuring impact. Even if expertly developed and superbly polished, software is worthless unless it has an impact in the hands of the end user. It is not enough to make users aware of a software's existence, they must be convinced the software they are currently using is inferior enough to endanger their work, and that the new software will remove that danger. Though, that is a difficult task in itself, as users must overcome their reluctance to modify their existing software stack.

The ultimate measure of impact stems from indications of usage. Ideally, it is best if impact measurements are easy to factor and objective. Some possible metrics include: growth of the contributor base; number of users; number of software releases; number of downloads and citations; level of user satisfaction; level of vendor adoption; number of research groups using the resources; percentage of reasonably resolved tickets; time-to-resolve tickets; number of publications citing or using the resource; and subjective user experience reports.

Calculating metrics for LAPACK, for example, we see there have been around 6.4 million downloads of LAPACK and 1.5 million downloads from ScaLAPACK per year, averaged over the last 29 years for LAPACK and over the last 25 years for ScaLAPACK.[31] This is for the packages as well as various components from the packages. These packages are also included in software products like Matlab, Julia, R, Mathematica, and Intel's MKL, which we cannot easily count. Indeed, many scientists are not even aware they are using LAPACK, let alone the BLAS.

As much of the scientific software stack is open source, one can also investigate different package managers (for example, Spack[19]) to measure dependencies and usage, or use sites that do this automatically (for example, libraries.io monitors close to five million open source packages across 37 different package managers). However, usage typically needs to be compared to other developments, quality and quantity is also important, and measurements become more difficult and subjective. Although there are several measures of impact that can be used for software, they are not well established nor supported, which stands in contrast to the number of citations or h-index calculated for publications.

Licensing for users and manufacturers. An important lesson learned for scientific software is the significance of its licensing. Much of the scientific software is open source, frequently using a Berkeley Software Distribution (BSD)-derived license, which originated in the BSD Unix OS.[29]

The BSD license is a permissive, free software, license imposing minimal restrictions on the use and redistribution of covered software. A BSD style license is a good choice for long duration research or other projects that require a development environment that has near zero cost for end users, will evolve over a long period of time, and permits anyone to retain the option of commercializing final results with minimal legal issues.

The success of the scientific software stack can, in part, be attributed to the choice of software licensing. Not only is the software, in general, of high quality, well tested, portable, and actively maintained; it is also capable of being incorporated into other software applications with minimal restrictions on the use and redistribution of the application software; in other words, the license is not a hindrance and allows users to employ the software how they see fit.

Funding for research and development. With the development of mathematical software, the process begins with a sound foundation in mathematics that expresses the correctness and stability of the computation. A numerical algorithm is then developed that expresses the mathematics as an algorithm that encompasses the various cases the mathematics considers. A more complete picture would be:

- the development and analysis of algorithms for standard mathematical problems which occur in a wide variety of applications;
- the practical implementation of mathematical algorithms on computing devices, including study of interactions with particular hardware and software systems;
- the environment for the construction of mathematical software, such as computer arithmetic systems, languages, and related software development tools;
- software design for mathematical computation systems, including user interfaces;
- testing and evaluation of mathematical software, including methodologies, tools, testbeds, and studies of particular systems; and,
- issues related to the dissemination and maintenance of software.

Each of these items requires an investment of time and funding to successfully accomplish its task. The National Science Foundation and the Department of Energy have contributed to the promotion of various aspects of this overall research and development process.

Personnel for long-running projects. Training and retention of a cadre of young people to engage in long term projects are critical. A strong research program cannot be established without a complementary education component, which is as important as adequate infrastructure support. A continuing supply of high-quality computational scientists available for work in our field is critical. This starts with graduate students, who contribute to the software development, and continues with post-docs who care about the development and help with the research directions, as well as research professors and colleagues, who contribute to the overall effort. Without a continuous effort full of qualified people at these levels, such long-term projects cannot be carried out at our universities. Students and post-docs are with the project for only a short time. It is critical that the design is well documented, and the documentation is faithful to the software that is developed. For the student, it can lead to a thesis or dissertation. For post-docs, it can solidify their interest in the field and lead to new research areas.

Traditionally, individual researchers working alone or in pairs have characterized the style of much of the work in the sciences. This situation is different in computational science where increasingly a multidisciplinary team approach is required. There are several compelling reasons for this. First and foremost, problems in modern scientific computing transcend the boundaries of a single discipline. In general, the computational approach has made science more interdisciplinary than ever before. There is a unity among the various steps of the overall modeling process from the formulation of a scientific or engineering problem to the construction of appropriate mathematical models, the design of suitable numerical methods, their computational implementation, and, last but not least, the validation and interpretation of the computed results. For most of today's complex scientific or technological computing problems a team approach is required involving scientists, engineers, applied and numerical mathematicians, statisticians, and computer scientists.

Clearly, the investment costs, as well as the longer duration of typical computational projects—especially when extensive software development is involved—necessitate a certain continuity and stability of the entire research infrastructure.

Conclusion

Advancing to the next stage of growth for computational simulation and modeling will require us to solve basic research problems in computer science and applied mathematics, at the same time as we create and promote a new paradigm for the development of scientific software. To make progress on both fronts simultaneously will require a level of sustained, interdisciplinary collaboration among the core research communities.

Existing numerical libraries will need to be rewritten and extended considering emerging architectural changes. The technology drivers will necessitate the redesign of existing libraries and will force re-engineering and implementation of new algorithms. Because of the enhanced levels of concurrency on future systems, algorithms will need to embrace asynchrony to generate the number of required independent operations.

As we enter an era of great change, strategic clarity and vision will be essential. Technology disruptions will also require innovative new ideas in mathematics and computer science. We need sustained investments in creative individuals and high-risk concepts.

The community has long struggled to settle on a good model for sustained support for key elements of the software ecosystem. This issue will become more acute as we move to exascale and beyond. The community needs to recognize that software is really a scientific facility that requires long-term investments in maintenance and support.

Acknowledgments. Thanks to Terry Moore, Mark Gates, Piotr Luszczek, Stanimire Tomov, Sven Hammarling, Natalie Beams, and Rob Anderson for their help with this article. C

References

1. Abdelfattah, A. et al. A survey of numerical linear algebra methods utilizing mixed-precision arithmetic. *The Intern. J. of High-Performance Computing Applications 35*, 4 (2021), 344–369; https://doi.org/10.1177/10943420211003313
2. Agullo, E. et al. *Harnessing Supercomputers with a Sequential Task-based Runtime System 13*, 9 (2014), 1–14.
3. Anderson, E. et al. LAPACK User's Guide, 3rd Edition. *Society for Industrial and Applied Mathematics*, Philadelphia, 1999.
4. Ballard, G., Demmel, J., Holtz, O., Schwartz, O. Minimizing communication in numerical linear algebra. *SIAM J. on Matrix Analysis and Applications 32*, 3 (2011), 866–901.
5. Bauer, M., Treichler, S., Slaughter, E., Aiken, A. Legion: Expressing locality and independence with logical regions. In *Intern. Conf. for High-Performance Computing*. Networking, Storage and Analysis, SC 2012; doi: 10.1109/SC.2012.71.
6. Bohr, M. A 30-year retrospective on Dennard's MOSFET scaling paper. *IEEE Solid-State Circuits Society Newsletter 12*, 1 (2007), 11–13; doi:10.1109/N-SSC.2007.4785534.
7. Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Hérault, T., Dongarra, J. PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability. *Computing in Science and Engineering 99* (2013), 1; https://hal.inria.fr/hal-00930217
8. Bueno, J. et al. Productive programming of GPU clusters with OmpSs. In *Proceedings of the 2012 IEEE 26th Intern. Parallel and Distributed Processing Symposium*, (2012), 557–568; doi:10.1109/IPDPS.2012.58.
9. Choi, Y., Dongarra, J., Pozo, R., Walker, D. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the 4th Symp. Frontiers of Massively Parallel Computations* (McLean, VA; Oct. 19–21, 1992), 120–127.
10. Demmel, J., Grigori, L., Hoemmen, M., Langou, J. Communication-optimal parallel and sequential QR and LU factorizations. *SIAM J. of Scientific Computing 34*, 1 (2012), A206–A239; doi:10.1137/080731992.
11. Dokulil, J., Sandrieser, M., Benkner, S. Implementing the Open Community Runtime for Shared-Memory and DistributedMemory Systems. In *Proceedings of the 24th Euromicro Intern. Conf. on Parallel, Distributed, and Network-Based Processing*, (2016), 364–368; doi:10.1109/PDP.2016.81.
12. Dongarra, J., Bunch, J., Moler, C., Stewart, G. LINPACK users' guide. *SIAM.* (Philadelphia, PA, 1979); doi:10.1137/1. 9781611971811.
13. Dongarra, J., Croz, J., Duff, I., Hammarling, S. Algorithm 679: A set of Level 3 Basic linear algebra subprograms. *ACM Trans. Mathematical Software 16* (1990), 1–17.
14. Dongarra, J., Croz, J., Duff, I., Hammarling, S. A set of Level 3 Basic linear algebra subprograms. *ACM Trans. Mathematical Software 16* (1990), 18–28.
15. Dongarra, J., Croz, J., Hammarling, S., Hanson, R. An extended set of FORTRAN Basic linear algebra subprograms. *ACM Trans. Mathematical Software 14*, 1–17.
16. Dongarra, J., Croz, J., Hammarling, S., Hanson, R. Algorithm 656: An extended set of FORTRAN Basic linear algebra subprograms. *ACM Trans. Mathematical Software 14* (1988), 18–32.
17. Dongarra, J., Gates, M., Luszczek, P., Tomov, S. Translational process: Mathematical software perspective. *J. of Computational Science 52* (2021), 101216. Case Studies in Translational Computer Science; https://www.sciencedirect.com/science/article/pii/S1877750320305160
18. Dongarra, J., van de Geijn, R. Two-dimensional basic linear algebra communication subprograms. Tech. Rep. *LAPACK Working Note 37.* Computer Science Department, University of Tennessee, Knoxville, TN (1991).
19. Gamblin, T., et al. The Spack package manager: bringing order to HPC software chaos. In *Proceedings of 2015 ACM Supercomputing.* J. Kern, J. S. Vetter (Eds.), 40:1–40:12; http://dblp.uni-trier.de/db/conf/sc/sc2015.html#GamblinLCLMSF15
20. Garbow, B., Boyle, J., Moler, C., Dongarra, J. Matrix eigensystem routines—EISPACK guide extension. LNCS 51. Springer, Berlin, 1977; doi:10.1007/3-540-08254-9.
21. Haidar, A., Kurzak, J., Luszczek, P. An improved parallel singular value algorithm and its implementation for multicore hardware. In *Proceedings of the Intern. Conf. on High Performance Computing. Networking, Storage and Analysis.* ACM, 2013, 90.
22. Haidar, A., Luszczek, P., Dongarra, J. New algorithm for computing eigenvectors of the symmetric eigenvalue problem. In *Proceedings of the Workshop on Parallel and Distributed Scientific and Engineering Computing.* (Phoenix, AZ, 2014), 130; doi:10.1109/IPDPSW.
23. Heller, T., Kaiser, H., Iglberger, K. Application of the ParalleX execution model to stencil-based problems. *Computer Science—R&D 28*, 2–3 (2013), 253–261; doi:10.1007/s00450-012-0217-1.
24. Kung, H., Leiserson, C. Systolic arrays (for VLSI). In *Proceedings of Society for Industrial and Applied Mathematics* (1978), 256–282.
25. Lawson, C., Hanson, R., Kincaid, D., Krogh, F. Basic linear algebra subprograms for FORTRAN usage. *ACM Trans. Math. Soft. 5* (1979) 308–323.
26. McCalpin, J. et al. Memory bandwidth and machine balance in current high-performance computers. *IEEE CS Tech. Committee on Computer Architecture Newsletter 2*, 19–25 (1995); https://www.cs.virginia.edu/stream/
27. Mohiyuddin, M., Hoemmen, M., Demmel, J., Yelick, K. Minimizing communication in sparse matrix solvers. In *Proceedings of the Conf. on High-Performance Computing Networking, Storage and Analysis.* ACM, 2009, 36.
28. OpenMP 5.0 Complete Specifications, Nov. 2018; https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification5.0.pdf.
29. Open-Source Initiative. The 3-clause BSD license, 1998; https://opensource.org/licenses/BSD-3-Clause.
30. Strohmaier, E., Meuer, H., Dongarra, J., Simon, H. The TOP500 list and progress in high-performance computing. *Computer 48*, 11 (2015), 42–49. doi:10.1109/MC.2015.338.
31. University of Tennessee. Oak Ridge National Laboratory. Netlib Libraries Access Counts; http://www.netlib.org/master_counts2.html
32. Whaley, R., Petitet, A., Dongarra, J. Automated empirical optimization of software and the atlas project. *Parallel Computing 27*, 11 (2000).
33. Wilkes, M., Wheeler, D., Gill, S. *The Preparation of Programs for an Electronic Digital Computer* (Charles Babbage Institute Reprint). The MIT Press, Cambridge, MA, 1984.
34. Wilkinson, J., Reinsch, C. *Linear Algebra, Vol. II of Handbook for Automatic Computation.* F.L. Bauer (Chief Ed), 1971.

Jack J. Dongarra is a University Distinguished Professor of Computer Science at the University of Tennessee, Knoxville, TN. He also holds appointments with Oak Ridge National Laboratory, USA, and the University of Manchester, U.K.