Translational Process: Mathematical Software Perspective

Jack Dongarra

University of Tennessee

Oak Ridge National Laboratory

University of Manchester

Mark Gates

University of Tennessee

Piotr Luszczek

University of Tennessee

Stanimire Tomov

University of Tennessee

Abstract

Each successive generation of computer architecture has brought new challenges to achieving high performance mathematical solvers, necessitating development and analysis of new algorithms, which are then embodied in software libraries. These libraries hide architectural details from applications, allowing them to achieve a level of portability across platforms from desktops to world-class high performance computing (HPC) systems. Thus there has been an informal translational computer science process of developing algorithms and distributing them in open source software libraries for adoption by applications and vendors. With the move to exascale, increasing intentionality about this process will benefit the long-term sustainability of the scientific software stack.

Keywords: dataflow scheduling runtimes, hardware accelerators, communication avoiding algorithms

Email addresses: `dongarra@icl.utk.edu` (Jack Dongarra), `mgates3@icl.utk.edu` (Mark Gates), `luszczek@icl.utk.edu` (Piotr Luszczek), `tomov@icl.utk.edu` (Stanimire Tomov)

URL: `http://www.netlib.org/utk/people/JackDongarra` (Jack Dongarra), `http://www.icl.utk.edu/~luszczek` (Piotr Luszczek)

1. Introduction

High-performance computers continue to increase in speed and capacity, with exascale machines expected to be delivered in 2021. 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 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 to multi-core architectures. Instead of getting improved performance for free, 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 heavy-weight CPU cores (using out-of-order execution, branch prediction, hardware prefetching, etc.) 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 has to explicitly program for multi-core CPUs and GPU 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 (Figure 1). 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 1–2 GHz, exascale machines, with 10^{18} floating point operations per second, must have billion-way parallelism. This is currently anticipated to be achieved by roughly 1.5 GHz $\times$ 10,000 nodes $\times$ 100,000 thread-level and vector-level parallelism. Thus 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

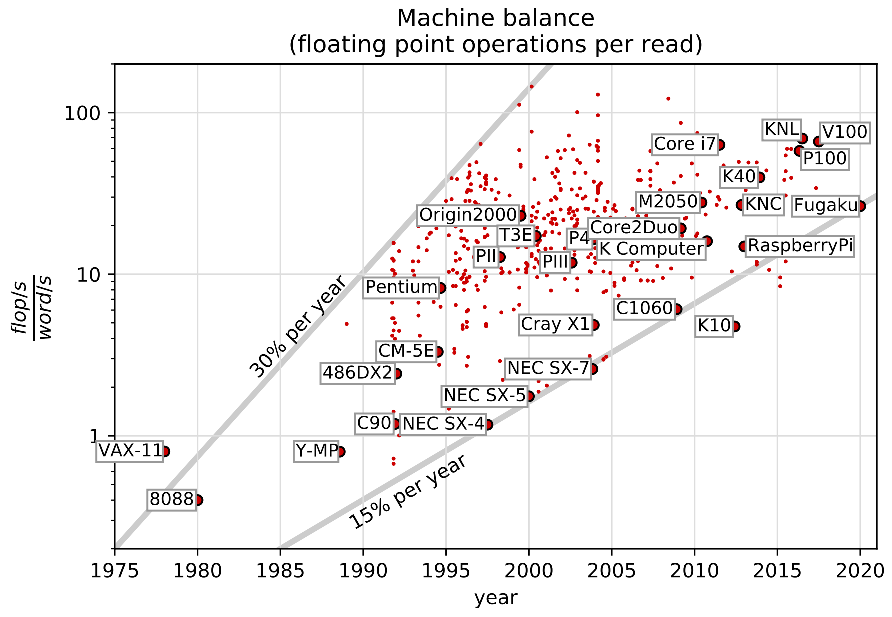

Figure 1: Processor and machine balance increasing, making communication relatively more expensive. Data from vendor specs and STREAM benchmark [1].

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); 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 of 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. But 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. At the same time, it seems prudent, if possible, to build on top of existing libraries instead of developing entirely new ones, since this will amortize some of the software maintenance costs, provide backward com-

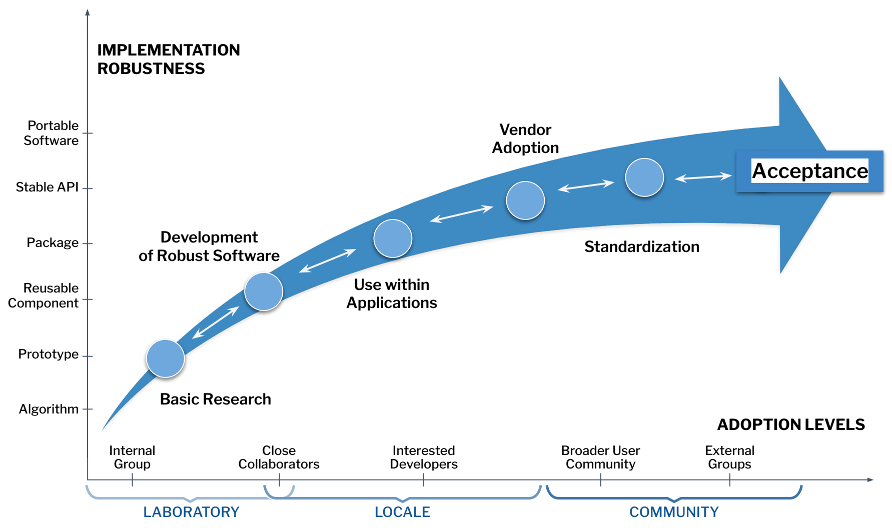

Figure 2: Translational approach for mathematical software.

patibility, and make transition for applications easier; and yet including radically different algorithms and methods at a low level, without radically altering usage characteristics of familiar packages at a high level, is a difficult software engineering problem. Moreover, many HPC applications will need to run on platforms ranging from leadership-class machines to smaller-scale clusters and workstations. 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.

This process of writing new generations of numerical software for new architectures has, informally, led to the translational process illustrated in Figure 2, which starts with *basic research* to develop high performance, numerically stable methods. This research grew out of a motivation to have efficient and stable algorithms on state-of-the-art architectures. Out of that research comes new mathematical algorithms that are developed into *robust software libraries* that are portable across platforms and include an extensive testing suite and documentation. *Applications* start to use these libraries, which are eventually *adopted by system vendors* such as AMD, Cray, IBM, and Intel for inclusion in their system software. Ideally, software goes through a *standardization process*, as in the case of MPI and BLAS, while other software becomes a de facto standard, like LAPACK. With this standardization comes *widespread acceptance*. Throughout this process, feedback is exchanged between the math library developers, application developers, and vendors. Underlying this process is an environment that includes: community involvement; an emphasis on high performance, efficiency, and portability; development of software that is freely available under a liberal open-source license; and ongoing software maintenance of the libraries. This general translational process was published by Abramson and Parashar [2]. Here, basic research and robust software corresponds to the lab in their concept; early adoption by applications and vendors corresponds to the locale, and standardization and widespread acceptance corresponds to the community. In this paper, we will look at how this translational research has affected the development of mathematical software libraries.

2. Background

Today's scientists often tackle problems that are too difficult to parse theoretically, or too difficult or dangerous 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?

At the application level, science must be captured in mathematical models, which are expressed algorithmically and ultimately encoded as software. Accordingly, much of the grant funding goes to support this modeling, which requires intimate collaboration among domain scientists, computer scientists, and applied mathematicians. This process relies on a large infrastructure of mathematical libraries, protocols, and system software that has taken years to build up and must be maintained, ported, and enhanced for many years to come in order to preserve and extend the value of the application codes that depend on it. 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, as well as the individuals who designed and developed it.

2.1. Standards

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. While single implementations have the advantage of rapid development and implementation, widely embraced standards (e.g., MPI, BLAS, IEEE floating point standards, and numerical libraries) are based on the experience of a wider community and are often required by application groups.

2.1.1. BLAS

Since the early days of HPC, the Level 1, Level 2, and Level 3 BLAS standards [3, 4, 5, 6, 7] 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.

The key to using a high-performance computer effectively is to avoid unnecessary memory movement, providing considerable motivation to devise algorithms to 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 [8]. 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 (e.g., 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, i.e., $O(n^2)$ data movement to $O(n^3)$ arithmetic operations. In addition, parallelism can be exploited in two ways: (1) operations on distinct blocks may be performed in parallel; and (2) within the operations on each block, scalar or vector operations may be performed in parallel.

2.1.2. Batched Basic Linear Algebra Subprograms (BBLAS)

On new hardware and with new algorithms, BLAS started showing its age as application developers expressed their compute needs in the form of multiple BLAS calls for relatively small problem sizes. Batched BLAS fills this performance need by, on one hand, abstracting away low-level details; and, on the other hand, extending the original interface to express the computational needs of the application in a hardware-friendly way [9, 10].

2.2. Software PACKs

Delivering specialized scientific software in the form of packages, such as EISPACK [11], LINPACK [12], LAPACK [8], ScaLAPACK [13], and others (see Appendix A), 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 they 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 the majority of machine-specific details yet allow automated adaptation to the user's platform of choice.

LAPACK [8] 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 portability across diverse parallel architectures. The BLAS that LAPACK heavily relies on provide a portable, efficient, and flexible standard for application programmers.

Maintaining scalability of parallel algorithms over a wide range of architectures and numerous processors will likely require the granularity of computation to be adjustable to suit the particular circumstances in which the software executes. Our approach to this problem is block algorithms with adjustable block sizes. In addition, a suite of algorithms may be required to deal with the full range of architectural diversity and processor multiplicity likely to be available.

2.3. 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 [14]. The programs were written in machine language, and certainly no thought was given to portability; to have a library at all was remarkable. 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 it is running on and make a dynamic decision on which path to take to successfully and efficiently execute on the underlying architecture. Applications use these numerical libraries, and it is these libraries we expect to be portable across different architectures.

2.4. Specific Techniques and Approaches

2.4.1. 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* [15]. In the ensuing decades, a large number of task-based runtime systems have been proposed and remain active [16, 17, 18, 19, 20, 21, 22] 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.

2.4.2. 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 wall clock 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 map better to the hardware and transfer less data across the modern memory hierarchy and on-node interconnects [23, 24]. 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 [25, 26], including CALU and CAQR factorization algorithms [27]. After basic research established their advantages, communication avoiding algorithms are now being integrated into various libraries such as LAPACK, MAGMA, ELPA, SLATE, and vendor libraries, continuing the translational process.

2.4.3. 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 much more efficiently. 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 couple of decades, these techniques mostly remained in a prototype state and rarely made it into production code. Recently, the US Department of Energy (DOE) Exascale Computing Project (ECP) has allocated resources to bring these techniques into production.

2.4.4. 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 time 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 a distinct feedback from application needs back to the development of new algorithms.

2.4.5. 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.

3. Translational Process and Moving Forward

Given the relatively small community of supercomputing researchers, international collaborations are particularly important. First and foremost, the magnitude of the technical challenges that new architectures and systems bring with them—and the corresponding sweep of changes required for HPC software infrastructure—are formidable. In terms of feasibility, the task of recreating this infrastructure to meet the new realities of advanced scientific computing is simply too large for any one country, or small consortium of countries, to undertake on its own. Second, the complex web of interdependencies and side effects that exist among the software components of advanced computing infrastructure means that making sweeping changes to this infrastructure will require a high degree of coordination and collaboration. Moreover, the HPC software infrastructure serves scientific communities that include global collaborations working on problems of global significance and leveraging resources in transnational configurations.

Historically, HPC software has been developed and maintained by national laboratories, universities, hardware vendors, and small, independent companies. Notably, though, an increasing amount of the software used in supercomputing is developed in an open-source model. Indeed, over the last 30 years, the open source community has provided much of the software infrastructure on which the world's HPC systems, ranging from supercomputers to campus clusters, have depended for their performance and productivity. It has invested billions of dollars and years of effort to build most of the key components, including math libraries (e.g., LAPACK [8] and PETSc [28]), low-level performance counter interfaces (e.g., PAPI [29, 30]), MPI, GNU tools, and many others.

Although the investments in these separate software elements have been tremendously valuable, a great deal of productivity has also been lost because of the lack of planning, coordination, and key integration of technologies necessary to make them work together smoothly and efficiently, both within individual HPC systems and between different systems. Open-source development within a single project can be coordinated by a repository gatekeeper and an email discussion list, but there is no global mechanism working across the community to identify critical holes in the overall software environment, spot opportunities for beneficial integration, or specify requirements for more careful coordination. It seems clear that this completely uncoordinated development model will not provide the software needed to support the unprecedented parallelism required for peta/exascale computation on millions of cores or the flexibility required to exploit new hardware models and features, such as transactional memory, speculative execution, and GPUs and other accelerators. What is needed is an international effort to coordinate research activities to gain more. However, such an effort is hard to manage and co-fund.

Moreover, the successful evolution and maintenance of complex software systems are critically dependent on institutional memory—that is, on the continuous involvement of the few key developers who understand the software design—and stability and continuity are essential to preserving institutional memory. Whatever support model is used, it should enable stable organizations with decades-long lifetimes to maintain and evolve the software.

In any case, experience shows that the creation of a new, high-quality software stack for scientific computing, one which can meet both the diverse requirements of future applications and the rigors of peta/exascale hardware architectures, will demand investment on an unprecedented scale. To avoid significant disruptions in critical research agendas, we need to leverage the collective resources of the global community. Even leaving the magnitude of the investment required aside, the software infrastructure that must be created is intended to serve a very broad spectrum of science and engineering communities, all of which are international in scope and need to leverage resources at a variety of scales.

4. Impact and Lessons Learned

4.1. 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, though that is a difficult task in itself, as users must overcome their reluctance to modify their existing software stack. They must be convinced that the software they are currently using is inferior enough to endanger their work, and that the new software will remove that danger.

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, and MKL, which we cannot easily count.

As much of the scientific software stack is open source, one can also look into different package managers (e.g., Spack [32]) to measure dependencies and usage, or use sites that do this automatically (e.g., libraries.io monitors close to 5 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 a number of 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.

A measure of impact that combines both objective and subjective measures can be obtained if we look into particular areas. For example, in the area of algorithms and numerical libraries for current and upcoming HPC hardware, a good example is the DOE's ECP effort, which is a large-scale development and deployment project for a comprehensive, integrated software stack and exascale hardware technology development and its translation into DOE mission-critical applications. ECP applications, and their associated exascale challenge problems, were reviewed by external experts and carefully selected in 2016 based on key DOE criteria, including: significance and requirement of exascale resources, alignment with DOE mission and strategic priorities, impact to both DOE and the broader community, and experience of the teams in leveraging HPC systems [33]. The software technologies in ECP and the number of ECP applications that depend on each technology are shown in Figure 3.

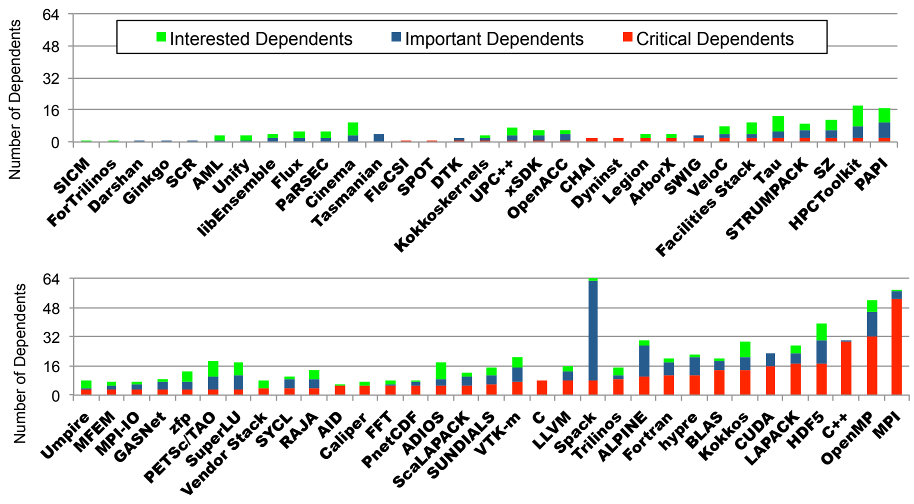

Figure 3: Number of application dependents for each of the software technologies in DOE's ECP project. There are a total of 64 applications and 64 software technologies in ECP.

There are 64 applications and 64 software technology projects in the ECP effort, ordered in Figure 3 on the x-axis by the number of applications that critically depend on them. Note, for example, that Spack [32] reaches the maximum of 64 dependencies, as all applications use Spack as their package manager to simplify and unify installation. After that, the software projects with the most dependents are the ones related to the programming model, namely: MPI, OpenMP, and C++ (in that order). The next projects with most critical dependents (17) are the LAPACK numerical library and the HDF5 open source file format for large, complex, heterogeneous data. Other notable software projects with a relatively large number of dependents are CUDA (programming for NVIDIA GPUs), Kokkos (portable programming model), BLAS, and ALPINE (scientific visualization).

4.2. Licensing for Users and Manufacturers

An important lesson learned for scientific software and its translation process 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. The modified or 3-clause BSD license states:

> Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
>
> (1) Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
>
> (2) Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
>
> (3) Neither the name of the Corporation nor the names of the contributors may be used to endorse or promote products derived from this software without specific prior written permission.

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.

4.3. 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 takes into account. 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;
- 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.

4.4. Personnel for Long Running Projects

Training and retention of a cadre of young people to engage in long term translational 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.

Unlike theoretical mathematics, computational mathematics, by its very nature, has a strong experimental component. As a result, research work proceeds in part in a laboratory mode similar to that in the experimental sciences. The laboratory equipment required for modern scientific computing ranges from local workstations to mainframe machines of various sizes, and supercomputers. This hardware is complemented by appropriate software systems and libraries.

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.

4.5. Roadblocks for ECP Translation Process

A major and valuable investment for a supercomputing ecosystem is its investment in people. The technology is maintained, exploited, and enhanced by the collective know-how of a relatively small cadre of supercomputing professionals—from those who design and build the hardware and system software to those who develop the algorithms and write the applications and programs. Their expertise is the product of years of experience. As supercomputing becomes a smaller fraction of research and development in information technology, there is a greater chance that those professionals will move out of supercomputing related employment and into more lucrative jobs. For example, their systems skills could be reused at Google, Facebook, or NVIDIA, and their algorithms skills would be useful on Wall Street.

5. Conclusions

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 promulgate 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 that requires a translational approach.

Existing numerical libraries will need to be rewritten and extended in light of 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.

Appendix A. PACKs over Decades

Project	Year	Authors
NATS Project	1971	Boyle, et al.
FUNPACK	1972	Cody
EISPACK	1972	Smith, et al.
RFK45	1977	Shampine & Watts
SLATEC	1977	DOE Community
LINPACK	1978	Dongarra, et al.
Level 1 BLAS	1978	Lawson, et al.
MINPACK	1979	More', et al.
DEPAC	1980	Shampine & Watts
DASSL	1982	Petzold
ODEPACK & SUNDIALS	1983	Hindmarsh
IEEE floating point	1985	Kahan et al.
Netlib	1985	Dongarra & Gross
Level 2 BLAS	1988	Dongarra et al.
PVM	1989	Geist, et al.
Level 3 BLAS	1990	Dongarra, et al.
PETSc	1991	Smith, et al.
ADIFOR	1992	Hovland, et al.
MPI	1992	Community
MPICH	1992	Gropp & Lusk
LAPACK	1992	Anderson, et al.
ScaLAPACK	1997	Blackford, et al.
SuperLU	1997	Li, et al.
Hypre	1998	Falgout, et al.
ARPACK	1998	Sorensen
ATLAS	2000	Whaley & Dongarra
PAPI	2000	Browne, et al.
Trilinos	2001	Heroux, et al.
Open MPI	2005	Community
IESP	2009	Community
PLASMA	2007	Kurzak, et al.
MAGMA	2009	Tomov, et al.
SLATE	2017	Gates, et al.

References

[1] J. D. McCalpin, et al., Memory bandwidth and machine balance in current high performance computers, IEEE computer society technical committee on computer architecture (TCCA) newsletter 2 (19–25) (1995).
URL https://www.cs.virginia.edu/stream/

[2] D. Abramson, M. Parashar, Translational research in computer science, Computer 52 (9) (2019) 16–23.

[3] C. L. Lawson, R. J. Hanson, D. Kincaid, F. T. Krogh, Basic linear algebra subprograms for FORTRAN usage, ACM Trans. Math. Soft. 5 (1979) 308–323.

[4] J. J. Dongarra, J. D. Croz, S. Hammarling, R. Hanson, An extended set of FORTRAN Basic Linear Algebra Subprograms, ACM Transactions on Mathematical Software 14 (1988) 1–17.

[5] J. J. Dongarra, J. D. Croz, S. Hammarling, R. Hanson, Algorithm 656: An extended set of FORTRAN Basic Linear Algebra Subprograms, ACM Transactions on Mathematical Software 14 (1988) 18–32.

[6] J. J. Dongarra, J. D. Croz, I. S. Duff, S. Hammarling, Algorithm 679: A set of Level 3 Basic Linear Algebra Subprograms, ACM Transactions on Mathematical Software 16 (1990) 1–17.

[7] J. J. Dongarra, J. D. Croz, I. S. Duff, S. Hammarling, A set of Level 3 Basic Linear Algebra Subprograms, ACM Transactions on Mathematical Software 16 (1990) 18–28.

[8] E. Anderson, Z. Bai, C. Bischof, S. L. Blackford, J. W. Demmel, J. J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. C. Sorensen, LAPACK User's Guide, Third Edition, Society for Industrial and Applied Mathematics, Philadelphia, 1999.

[9] J. Dongarra, I. Duff, M. Gates, A. Haidar, S. Hammarling, N. J. Higham, J. Hogg, P. Valero-Lara, S. D. Relton, S. Tomov, M. Zounon, A proposed API for Batched Basic Linear Algebra Subprograms, MIMS EPrint 2016.25, Manchester Institute for Mathematical Sciences, The University of Manchester, UK (Apr. 2016).
URL http://eprints.ma.man.ac.uk/2464/

[10] J. Dongarra, I. Duff, M. Gates, A. Haidar, S. Hammarling, N. J. Higham, J. Hogg, P. V. Lara, P. Luszczek, M. Zounon, S. D. Relton, S. Tomov, T. Costa, S. Knepper, Batched blas (basic linear algebra subprograms) 2018 specification (2018-07 2018).

[11] B. S. Garbow, J. M. Boyle, C. B. Moler, J. Dongarra, Matrix eigensystem routines – EISPACK guide extension, Vol. 51 of Lecture Notes in Computer Science, Springer, Berlin, 1977. doi:10.1007/3-540-08254-9.

[12] J. Dongarra, J. R. Bunch, C. B. Moler, G. W. Stewart, LINPACK users' guide, SIAM, Philadelphia, 1979. doi:10.1137/1.9781611971811.

[13] Y. Choi, J. J. Dongarra, R. Pozo, D. W. Walker, ScaLAPACK: a scalable linear algebra library for distributed memory concurrent computers, in: Proceedings of the fourth symposium on the frontiers of massively parallel computation (Frontiers '92), McLean, Virginia, Oct 19–21, 1992, 1992, pp. 120–127.

[14] M. V. Wilkes, D. J. Wheeler, S. Gill, The Preparation of Programs for an Electronic Digital Computer (Charles Babbage Institute Reprint), The MIT Press, 1984.

[15] H. T. Kung, C. E. Leiserson, Systolic arrays (for VLSI), in: Sparse Matrix Proceedings, Society for Industrial and Applied Mathematics, 1978, pp. 256–282, ISBN: 0898711606.

[16] M. Bauer, S. Treichler, E. Slaughter, A. Aiken, Legion: Expressing locality and independence with logical regions, in: International Conference for High Performance Computing, Networking, Storage and Analysis, SC, 2012. doi:10.1109/SC.2012.71.

[17] E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, S. Thibault, Harnessing Supercomputers with a Sequential Task-based Runtime System 13 (9) (2014) 1–14.

[18] T. Heller, H. Kaiser, K. Iglberger, Application of the ParalleX execution model to stencil-based problems, Computer Science - Research and Development 28 (2-3) (2013) 253–261. doi:10.1007/s00450-012-0217-1.

[19] J. Dokulil, M. Sandrieser, S. Benkner, Implementing the Open Community Runtime for Shared-Memory and Distributed-Memory Systems, Proceedings - 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2016 (2016) 364–368doi:10.1109/PDP.2016.81.

[20] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguadé, J. Labarta, Productive programming of GPU clusters with OmpSs, Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium, IPDPS 2012 (2012) 557–568doi:10.1109/IPDPS.2012.58.

[21] OpenMP 5.0 Complete Specifications, https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf (Nov 2018).

[22] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, J. Dongarra, PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability, Computing in Science and Engineering 99 (2013) 1. doi:10.1109/MCSE.2013.98.
URL http://hal.inria.fr/hal-00930217

[23] A. Haidar, P. Luszczek, J. Dongarra, New algorithm for computing eigenvectors of the symmetric eigenvalue problem, in: Workshop on Parallel and Distributed Scientific and Engineering Computing, IPDPS 2014 (Best Paper), IEEE, IEEE, Phoenix, AZ, 2014. doi:10.1109/IPDPSW.2014.130.

[24] A. Haidar, J. Kurzak, P. Luszczek, An improved parallel singular value algorithm and its implementation for multicore hardware, in: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, ACM, 2013, p. 90.

[25] G. Ballard, J. Demmel, O. Holtz, O. Schwartz, Minimizing communication in numerical linear algebra, SIAM Journal on Matrix Analysis and Applications 32 (3) (2011) 866–901.

[26] M. Mohiyuddin, M. Hoemmen, J. Demmel, K. Yelick, Minimizing communication in sparse matrix solvers, in: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, ACM, 2009, p. 36.

[27] J. Demmel, L. Grigori, M. Hoemmen, J. Langou, Communication-optimal parallel and sequential QR and LU factorizations, SIAM Journal of Scientific Computing 34 (1) (2012) A206–A239. doi:10.1137/080731992.

[28] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. D. Gropp, D. Karpeyev, D. Kaushik, M. G. Knepley, D. A. May, L. C. McInnes, R. T. Mills, T. Munson, K. Rupp, P. Sanan, B. F. Smith, S. Zampini, H. Zhang, H. Zhang, PETSc users manual, Tech. Rep. ANL-95/11 - Revision 3.13, Argonne National Laboratory (2020).
URL https://www.mcs.anl.gov/petsc

[29] H. Jagode, A. Danalis, H. Anzt, J. Dongarra, PAPI software-defined events for in-depth performance analysis, The International Journal of High Performance Computing Applications 33 (6) (2019) 1113–1127.

[30] A. Danalis, H. Jagode, T. Herault, P. Luszczek, J. Dongarra, Software-defined events through PAPI, in: 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Rio de Janeiro, Brazil, Brazil, 2019, pp. 363–372, dOI: 10.1109/IPDPSW.2019.00069.

[31] University of Tennessee, Oak Ridge National Laboratory, Netlib Libraries Access Counts.
URL http://www.netlib.org/master_counts2.html

[32] T. Gamblin, M. P. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, S. Futral, The Spack package manager: bringing order to HPC software chaos., in: J. Kern, J. S. Vetter (Eds.), SC, ACM, 2015, pp. 40:1–40:12.
URL http://dblp.uni-trier.de/db/conf/sc/sc2015.html#GamblinLCLMSF15

[33] F. Alexander, A. Almgren, J. Bell, A. Bhattacharjee, J. Chen, P. Colella, D. Daniel, J. DeSlippe, L. Diachin, E. Draeger, A. Dubey, T. Dunning, T. Evans, I. Foster, M. Francois, T. Germann, M. Gordon, S. Habib, M. Halappanavar, S. Hamilton, W. Hart, Z. Huang, A. Hungerford, D. Kasen, P. R. C. Kent, T. Kolev, D. B. Kothe, A. Kronfeld, Y. Luo, P. Mackenzie, D. McCallen, B. Messer, S. Mniszewski, C. Oehmen, A. Perazzo, D. Perez, D. Richards, W. J. Rider, R. Rieben, K. Roche, A. Siegel, M. Sprague, C. Steefel, R. Stevens, M. Syamlal, M. Taylor, J. Turner, J.-L. Vay, A. F. Voter, T. L. Windus, K. Yelick, Exascale applications: skin in the game, Philosophical Transactions of the Royal Society. A, Mathematical, Physical and Engineering Sciences 378 (2166) (1 2020). doi:10.1098/rsta.2019.0056.