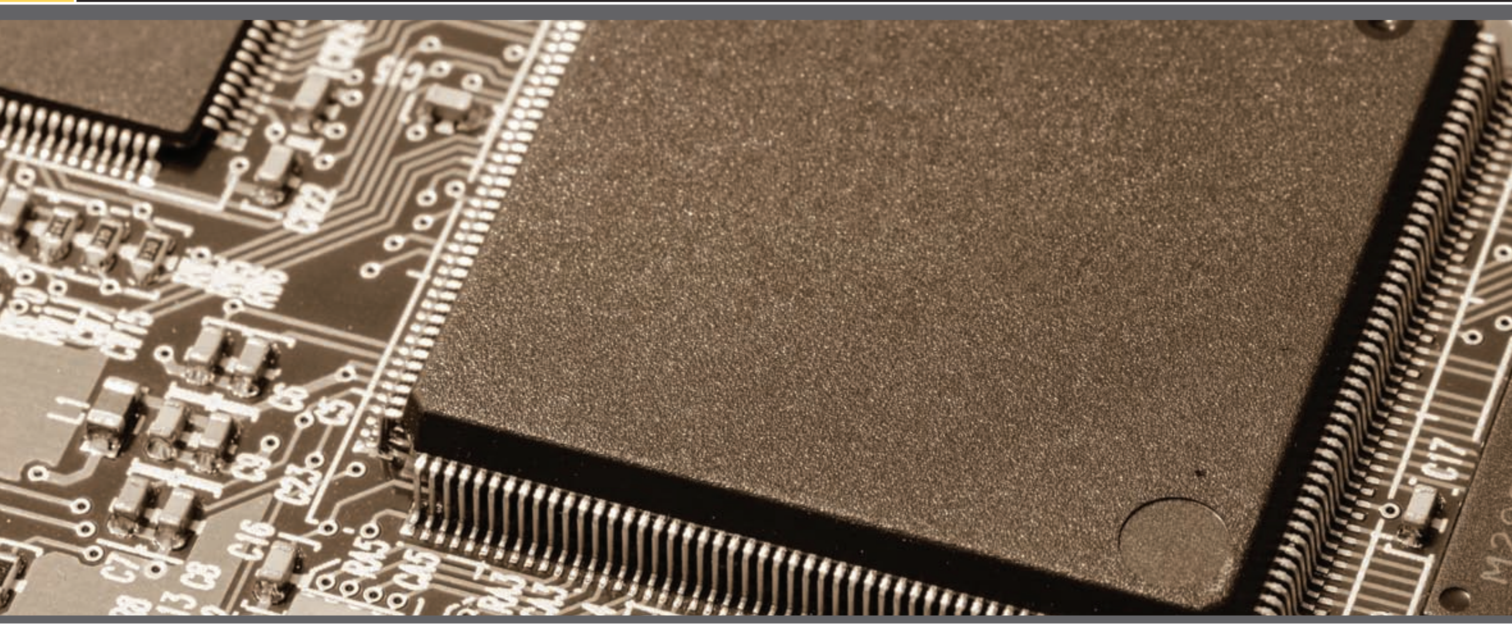

With Extreme Computing, the Rules Have Changed

Jack Dongarra, Stanimire Tomov, Piotr Luszczek, Jakub Kurzak, Mark Gates, Ichitaro Yamazaki, Hartwig Anzt, Azzam Haidar, and Ahmad Abdelfattah | University of Tennessee Knoxville

On the eve of exascale computing, traditional wisdom no longer applies: high-performance computing is gone as we know it. A range of new algorithmic techniques is emerging in the context of exascale computing, many of which defy the common wisdom of HPC and are considered unorthodox but could turn out to be a necessity in the near future.

Science priorities lead to scientific models, and models are implemented in the form of algorithms. Algorithm selection is based on various criteria, such as accuracy, verification, convergence, performance, parallelism, and scalability. Models and associated algorithms aren't selected in isolation but must be evaluated in the context of the existing computer hardware environment. Algorithms that perform well on one type of computer hardware could become obsolete on newer hardware, so selections must be made carefully and may change over time.

Future trends in high-performance computing (HPC) can be seen in the US Department of Energy CORAL (Collaboration of Oak Ridge, Argonne, and Livermore) initiative, which is aimed at deploying "pre-exascale" machines with performance in the range of hundreds of petaFLOPS. The list of challenges facing the algorithm developer includes targeting massive parallelism, reaching millions of cores, dealing with hybrid architectures (including GPU accelerators), and managing complex memory hierarchies, all with smaller amounts of faster 3D-stacked memories and larger amounts of traditional memories.

In the sections to follow, we first describe the particular challenges posed by new hardware technologies and then present a selection of emerging, often unorthodox, algorithmic techniques to deal with these challenges.

Moore's Law and Dennard Scaling

Moore's law[1] is the most often quoted dictum by which the silicon revolution took over our lives. Never mind that it was meant to last only until 1975 and was later adjusted to fit a slower (but still exponential) growth rate.[2] In essence, the law observes exponential growth of transistors in every new generation of integrated circuits and projects it into the following decade. While Gordon Moore himself suggested that this would allow higher speed for the same power (per unit area), it was Robert Dennard and colleagues[3] who showed how the observed speed of the chips would increase exponentially. Confluence of these trends gave us the proverbial "free lunch": write any code and wait for the Moore/Dennard duo to speed it up. The joyride ended abruptly around 2005, when the per-square-inch heat emitted by chips was on track to exceed that of a rocket nozzle—and soon after, the surface of the sun—due to the effect of increasing main-clock frequency.[4]

This ended Dennard scaling, but Moore's law progression continues, albeit in a somewhat abated form: the shrink periods in chip feature size are ever longer, each one requiring an even bigger investment in terms of science, innovation, engineering, and money. Instead of simply faster processors, we get processors with more cores. If we don't get faster results on the new generation of hardware, we can now blame only our coding skills: we didn't provide for the increased parallelism. Going beyond the outgoing breed of commercial 22-nm chips and the established 14-nm technology, IBM announced a successful research result by producing a 7-nm germanium integrated circuit.[5] Following suit, Intel plans to move away from silicon for the 7-nm manufacturing process.[6,7] Nobody is really surprised, because if the feature size shrinks by half in two years, then we have only about a decade before we reach the lattice parameter for silicon, which is 0.543 nm.

Clock Variation

For a long time, CPU speed has been associated with frequency.[8] But that trend has now gained a new meaning due to the flexible treatment of clock frequency in modern CPUs and GPUs. Frequency scaling originated in laptop CPUs; desktop and server CPUs, on the other hand, featured technologies such as Turbo Boost and Turbo Core. All of them allowed CPU frequency to be adjusted by external factors—usually, system load and available battery charge. The constant increases in clock frequencies came to an end when Dennard scaling no longer continued. The changes in CPU/GPU frequency continue, however, due to various factors. High frequency still bodes well for performance, and modern chips are capable of recognizing their workload to the point that when the demand for computing capacity increases, the clock revs up to complete the task at hand.

But there's a limit to this technique because the thermal bounds can't be breached without the chip suffering irreversible damage. So, as soon as the temperature sensors detect dangerous levels of dissipated heat, the frequency is scaled appropriately, regardless of workload status. A less obvious application for this technique pertains to regulation of the consumed power. It's hard to imagine why a processor would need to regulate its power consumption when thinking small. At extreme scales, the number of processing cores already reaches over a million cores. A sudden increase in application workload is magnified accordingly, which results in a sudden surge of demand for electric power unlike anything that power houses have ever faced. The usual fluctuations in electric power consumption have to do with time of day and are a result of changes in human activity. Over decades, these could be accounted for, although blackouts can still occur when too many people turn on their air conditioning in a short window of time. In comparison, CPUs and GPUs can reduce their power consumption when idle by nearly 90 percent, from a few hundred watts to a few tens. If only 10,000 processors changed their state from idle to fully loaded, the electrical power demand would change by nearly 1 MW, and not just for a particular time of day, as is the case for rush-hour traffic and evening TV watching.

Power houses can't cope with such swings in demand and heavily penalize offending sites, and supercomputer operators institute policies that help manage workload variation and associated power fluctuations, often through power capping and frequency adjustments. It then becomes hard to precisely answer the question, how fast is a CPU or GPU in my supercomputer? It depends on the temperature—when machine room cooling works well, processors can increase their frequency beyond the base level and thus magnify the entire supercomputer's performance. But if the electrical power is at a premium and the center can't draw any more from the power house, power capping will be in effect, and thus frequency will stay at the baseline level or even drop below it to counteract a potential surge and ensuing penalties. In short, the clock variation is here to stay, and as a consequence, not all cycles are made equal.

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 used to apply when every cycle and every instruction were 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 1,000 GFLOPs in double precision and bandwidth topping at 200 Gbytes/s. An order of magnitude more operations have to 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 anything else, and especially to the main memory that holds the majority of 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 1 Tbytes/s of bandwidth, compute devices will eventually achieve performance levels in excess of 10 TFLOPs, 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 have to look critically at the kind of operations that are required. And above all, we have to focus on data movement, synchronization points, and understanding the nature of 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 have to examine the amount of data that the algorithm accesses and choose a different one that can do away with fewer accesses—we call this *access-averse methodology*. We already have meaningful examples of a substantial payoff from this new approach throughout the numerical linear algebra field. We see it in eigenvalue and singular value computations, where algorithms with double the amount of floating-point work are already faster on current systems and will only get faster moving forward with technological advances in hardware. We also see this in sparse matrix computations that use subspace projection and improve accuracy through extra work while being overall faster than any other contending algorithm. While in some way it might be burdensome to rewrite old software and reinvent old algorithms, we ultimately stand to benefit from the new approaches that technological progress requires us to invent.

Responsibly Reckless

The ultimate goal for high-performance numerical libraries, which has also driven their progress over the years, is to deliver accurate results in the fastest possible time. Accuracy by itself, however, is often associated with extra computational effort and therefore is at odds with speed. Maintaining an accuracy versus speed standoff for traditional algorithms on today's and tomorrow's emerging extreme computing systems is becoming increasingly challenging. Indeed, applications from big data analytics to machine learning, in which "sensors" produce extreme amounts of data (including redundant or faulty data), require various optimizations to sift through and find a "best" solution in a limited time period. However, this push also motivates the development of algorithms that we call *responsibly reckless*, algorithms that compute quickly while still being responsible for accuracy through nontraditional, innovative approaches.

Examples are numerous, including mixed-precision and randomization algorithms, as well as some of the already mentioned methods that tradeoff more flops to gain in overall time to solution. To illustrate the concept, let's concentrate on the general enough problem of solving a linear system of equations:

$$Ax = b.$$

The traditional approach is to use a Gaussian elimination (GE) with partial pivoting, which in matrix form is known as LU factorization with partial pivoting. In this approach, A is decomposed so that $PA = LU$, where L and U are lower and upper triangular matrices, respectively, and P is a permutation matrix. This is achieved by reducing

A, column by column, to the upper triangular U using elementary transformations (swapping rows, scaling a row, and adding a scaled row to another row) that form the lower triangular L and the permutation P matrices. To reduce a current column to zeros below the diagonal, we can use the row going through the diagonal—this results in an LU factorization without pivoting, which is numerically unstable. With partial pivoting we select the row that has the largest absolute value in the current column, swap it with the diagonal row, and use it in the reduction. This strategy is sufficient in practice to adequately reduce round-off error. If A is of size $n \times n$, the computation is of order $O(n^3)$ and the pivoting adds an $O(n^2)$ complexity. While this is a lower-order term, the challenges and communication overhead due to pivoting have become critical on new architectures, especially on GPUs in variations when A is symmetric and indefinite.[9]

GE stability is strongly related to the growth factor that measures how large the entries of the matrix become in the GE process. As in many numerical algorithms, the choice of a pivoting strategy is the result of a tradeoff between stability concerns and performance. In respect to this, a good GE algorithm should minimize the growth factor (to provide numerical stability) and the amount of pivoting (to avoid penalizing performance). An example of a responsibly reckless approach here is based on a randomization technique where the original matrix A is transformed/preconditioned into a sufficiently "random" matrix so that, with a probability close to 1, pivoting isn't needed. The technique was initially described by Douglas Parker[10] and extended later for dense linear systems, either general[11] or symmetric indefinite.[9] The randomization is referred to as random butterfly transformation (RBT)[11] and consists of a multiplicative preconditioning U^TAV, where U and V are chosen among a particular class of random matrices called recursive butterfly matrices. Then, GE with no pivoting is performed on the matrix U^TAV and, to solve $Ax = b$, we solve $(U^TAV)y = U^Tb$, followed by $x = Vy$.

Bombardment

With the growing level of parallelism in computer architectures, global reductions are increasingly becoming the performance-limiting factor. At the same time, the number of (parallel) floating-point operations in an algorithm can be expected to become less relevant for the execution time, which promotes the idea of adding extra operations in favor of improved algorithm properties. Following this idea, the concept of algorithmic bombardment[12] was proposed nearly 20 years ago for the iterative solution of large, sparse linear systems.

The traditional approach for the iterative solution process is to select one iterative method and apply it to the problem. In particular, Krylov subspace solvers have worked well for a large range of problems, but for a problem with unknown characteristics, it's often difficult to identify the method of choice. The large variety of Krylov solvers to choose from also makes the selection difficult. The distinct methods differ in how they deal with the initial guess the solver is started with, as well as the linear problem's characteristics, such as eigenvalue distribution. Depending on this, an iterative solver can converge faster or slower—or in the worst case, even break down.

The concept of algorithmic bombardment addresses this challenge in a sledgehammer fashion: instead of addressing the problem with a specific iteration method, a poly-algorithm attack applies a preselected set of solvers simultaneously. The key idea is to successively drop the methods that break down and benefit from the fast convergence of the most suitable method included in the set. Although the poly-iterative approach has some overhead, it has shown to be very efficient, in particular for a set of similar Krylov-based methods. The structural similarity of these solvers helps keep a low number of synchronization points: the central (and computationally most expensive) building block of all Krylov methods is a sparse matrix vector product needed for the generation of the Krylov subspace. Algorithms are also composed of inherently parallel vector updates and global reduction operations that require synchronization. Multiplying the sparse matrix with a set of vectors instead of a single vector usually incurs little computational overhead, with no additional synchronizations. Similarly, the simultaneous reduction of multiple vectors is insignificantly more expensive than a single reduction on parallel architectures. Finally, the update of a set of vectors being part of the distinct algorithms might not incur any overhead, as it can often be hidden with the global reduction phases guiding the algorithm's execution time.

Consequently, interleaving a set of Krylov methods by generating the distinct subspaces via a blocked sparse matrix vector product—and gathering the reduction phases such that a low number of synchronization points is maintained—typically results in a small runtime overhead compared to running only one iterative solver. But while the outcome when choosing one method is unknown, the poly-iterative approach not only increases the chance of one method converging but also provides the best convergence rate of the solvers included in the set.

The poly-iterative approach offers additional benefits for fault-tolerance aspects. The increasing complexity of hardware platforms undoubtedly results in a higher rate of soft errors such as bit flips. These can degrade an iterative solver's convergence or, in the worst case, even result in the breakdown of an algorithm. For the algorithmic bombardment approach, it's less crucial if one method of the set breaks down due to numerical instability or soft-error: the respective solver is simply dropped, and the iterations continue with one less solver. Obviously, this doesn't make the poly-iterative approach bit-flip proof, but it enhances the chance of successful termination compared to a single iteration method.

Dataflow Scheduling

Dataflow scheduling is a very old idea that offers numerous benefits to traditional multithreading and message passing, but it somehow never really went mainstream. Until now. Part of the problem is cultural, as dataflow programming takes some control away from the programmer by forcing a more declarative, rather than imperative, style of coding. You need to let the system decide what work is executed where and at what time. With the prospect of billion-way parallelism looming on the HPC horizon, there simply is no other choice.

The multicore revolution of the 2000s brought the idea back and put it in the mainstream. One of the first task-based multithreading systems that received attention in multicore times was the Cilk language, originally developed at MIT in the 1990s and mostly offering nested parallelism, heavily geared toward recursive algorithms. About the same time, the idea of superscalar scheduling gained traction, based on scheduling tasks by resolving data hazards in real time, in a similar way that superscalar processors dynamically schedule instructions.

The appeal of the superscalar model is its simplicity. You basically present the compiler with serial code, where tasks are, in principle, functions. They have to be side-effect-free (no accesses to global variables and so on). Then you have to mark the parameters as input, output, or in-out. Based on this information, at runtime, the task graph is built, and tasks are scheduled in parallel to multiple cores by resolving data dependencies and hazards: read after write, write after read, write after write. This is a powerful idea that's simple to grasp and allows for scheduling complex workloads to many cores.

This technique was pioneered by a project from the Barcelona Supercomputer Center, which went through multiple names as its hardware target was changing: GridSs, CellSs, SMPSs, OMPSs, and StarSs, where "Ss" stands for superscalar.[13] A similar development was the StarPU project from INRIA, which applied the same methodology to systems with GPU accelerators. It was named for its capability to schedule work to CPUs and GPUs, therefore *PU. One scheduler (SuerGlue) was developed at Uppsala University, and we developed our own (Quark) at the University of Tennessee Knoxville (UTK), implementing the Plasma numerical library on top of it. At some point, all these projects received extensions for scheduling in distributed memory.

Nothing accelerates adoption of a new paradigm more than standardization. Luckily, the OpenMP community has been swiftly moving forward with standardization of new scheduling techniques for multicores. First, the OpenMP 3.0 standard adopted the Cilk scheduling model, then the OpenMP 4.0 standard adopted the superscalar scheduling model.[14] Not without significance is the fact that the GNU compiler suite was also quick to follow with high-quality implementations of the new extensions. The results of our initial investigation indicate that at this point there are no roadblocks to fully migrate the Plasma library from Quark to OpenMP, which, in our minds, speaks highly of the standard, as Plasma is a fully featured numerical library.

Sadly, the situation is worse in programming distributed-memory machines, where the MPI+X model is currently the predominant solution (meaning MPI plus OpenMP, POSIX threads, CUDA, OpenCL, OpenACC, and so on). While the superscalar model can be extended to support distributed memory, the paradigm is inherently nonscalable to very large core numbers, due to the serial bottleneck of unrolling the task graph at runtime. Numerous projects have tackled the problem, including Charm++ from the University of Illinois, Urbana-Champaign, Swift from Argonne, ParalleX from Louisiana State University (now Indiana State University), just to name a few, and we joined the fight with the PaRSEC and Pulsar systems developed at UTK. However, while dataflow scheduling seems inevitable at exascale, there's currently no paradigm on the radar that could serve as a basis for standardization.

Communication Avoiding

The cost of executing software can be modeled as a function of its computation and communication costs (such as in terms of required cycle time or energy consumption). For instance, the computation cost could be modeled based on the number of required FLOPS, while the communication could include synchronization and data transfer between parallel processing units, as well as data movement through levels of the local memory hierarchy. On modern computers, communication has become significantly more expensive compared to computation. To address this hardware trend, several communication-avoiding (CA) algorithms have been developed by redesigning existing algorithms to obtain the minimum communication cost for solving a particular problem.[15,16]

For example, significant efforts have been made to redesign dense matrix factorization algorithms to obtain the minimum communication costs, such as LU and QR factorization algorithms,[17] with traditional algorithms communicating to factorize each column of the dense matrix. In contrast, to reduce the number of required messages in a parallel environment via a CA algorithm, each parallel processing unit would factorize its local submatrix, followed by a global reduction of the local factors to compute the final factorization. A similar idea can be applied to reduce the data movement through the local memory hierarchy by splitting the matrix into submatrices that would fit in the faster memory. These algorithms typically increase the computational costs and numerical bounds by a small factor (such as a factor of log(*np*), where *np* is the number of processing units), but they can greatly improve performance when communication dominates the costs of computing the factorization.

There have also been significant efforts[18] to avoid communication in the iterative subspace projection methods for solving sparse matrix problems (such as solving a linear system of equations or an eigenvalue or singular value problem). At each iteration of a traditional algorithm, a new basis vector of the projection subspace is generated, which requires communication. To reduce communication, CA algorithms generate a set of *s* basis vectors at once. Although the total computation or the communication volume of the iterations can increase depending on the surface-to-volume ratio of the local sparse submatrix, CA algorithms reduce the communication latency by a factor of *s*. Special care must be taken to maintain the solver's numerical stability, but emerging efforts[18] aim to improve the robustness of CA algorithms in practice (such as deflation and preconditioning to improve the convergence rates).

Mixed Precision

Traditionally, most scientific and engineering computing has been done in double-precision 64-bit arithmetic, with the expected ratio of single-precision (32-bit) to double-precision performance at 2× in favor of single. In the mid-2000s, interest in using lower precision spiked due to infiltration of the HPC domain by devices meant for consumer electronics and embedded systems, which offered outstanding performance in single precision and inferior performance in double. Along came the realization that, for some applications, single-precision accuracy is more than enough, while for others, the bulk of the work can be done in single precision, and lost digits can be cheaply restored by applying precision recovery techniques. Now that hybrid architectures, including manycore coprocessors such as the Intel Xeon Phi, and hardware accelerators, such as GPUs from Nvidia and AMD, are an integral part of the HPC landscape, it only seems natural to reap the benefits of mixed-precision computing.

The turmoil started with the Sony/Toshiba/IBM Cell processor, which initially offered 14× more performance in single precision than in double precision (later improved to 2× in the PowerXCell architecture). When the Cell lost momentum, GPU computing gained traction, with Nvidia leading the charge. In Nvidia products, native support for double precision started with the Tesla microarchitecture, and the performance ratio fluctuated from architecture to architecture, being 8:1 in Tesla, 2:1 in Fermi, and 3:1 in Kepler. The current Maxwell line of cards is likely to never receive fast double-precision units and will simply be bypassed for HPC applications in favor of moving on to Pascal, which comes with another twist. While its double-precision specs aren't yet known, it will provide very fast half-precision (16-bit) operations for applications in deep learning. In the meantime, AMD followed suit and introduced double precision in the R600 architecture. For AMD chips, the single-to-double performance ratio has been hovering consistently around 5:1, going as high as 4:1 in the Tahiti chips.

Some numerical computing applications are so well conditioned that single precision is all they need. A prominent example is wave scattering for oil exploration and computing radar signatures. Computational lithography is another example. Successful efforts have been made to explore single precision in harder problems, such as molecular dynamics simulations. In this case, a class of problems can be solved by mixing different precisions, such that most of the work is done in lower or faster precision, while a smaller amount of work is done in higher or slower precision, to recover the lost digits. Well-conditioned systems of linear equations can be solved in this manner by using the technique of iterative refinement.[19]

Hardware-supported higher precision can easily be emulated in software—for instance, a quadruple-precision number can be represented by two double-precision numbers, one storing the exponent and the higher-order bits of the mantissa, and the other storing the lower-order bits of the mantissa. While the exponent is limited to the double-precision exponent, a quadruple-precision mantissa can be represented. The cost is on the order of 10 lower-precision instructions to implement a single higher-precision instruction, but when applied judiciously to some parts of the algorithm, the technique opens the door for lowering precision in other parts of the algorithm.[20]

Future state-of-the-art numerical software should be able mix and match different precisions in different stages to achieve the required accuracy while maximizing performance.

Reproducibility

Exact reproducibility of computed results is a desirable feature and a common requirement of scientific software. It aides development to be able to make certain changes in the code without altering the end result. The landscape of extreme computing, with extreme levels of parallelism and dynamic scheduling, makes exact reproducibility prohibitively expensive.

One fundamental cause of nonreproducibility is that floating-point arithmetic isn't associative, so in general,

$$(a + b) + c \neq a + (b + c),$$

but rather there's a small round-off difference between the two results. Any time there's a similar reduction—such as a dot product or norm—runtime choices affect the results.[21] Using a different number of processors will cause a different grouping, yielding a small difference in the result. Tuning the block size for optimal performance, for example, affects the result. Because of SIMD vector units in today's processors, even changing the memory alignment of data can change how data is associated.

Moving a computation from a CPU to an accelerator, such as a GPU or Xeon Phi, likewise changes results as the computation is reorganized to best fit the hardware characteristics. Dynamically scheduled algorithms could choose where to do a computation—on a CPU or on an accelerator—based on runtime load balancing.

Often, these cause only small changes in the values. When a decision is based on intermediate values, however, the end result can be significantly different. In LU decomposition, for instance, the largest element in a column is chosen as the pivot. If two elements are close in magnitude, a small perturbation can cause a different pivot to be chosen, so the resulting permutation P and LU factors are completely different, while still satisfying $PA = LU$. When used to solve a system, $Ax = b$, the result is still within some tolerance of the correct x. A similar phenomenon occurs in the sign choice when computing Householder reflectors, hence affecting QR decomposition, eigenvalue, and singular-value problems.

Another source of nonreproducibility is application-based fault tolerance (ABFT),[22] which incurs less overhead compared to traditional checkpoint-restart fault tolerance. While checkpoint-restart restores the exact contents of memory after a fault, ABFT recomputes missing data based on checksums. The recomputation inherently incurs round-off errors different than the original computation.

Some algorithms also include random data, such as a random initial starting guess in iterative algorithms, random sampling in Monte Carlo methods, or RBT. Although repeatable pseudorandom generators produce reproducible results for debugging, it isn't practical or desirable to always use the same pseudorandom sequence in production runs.

Some of these changes can be controlled, for instance, by ensuring that data is aligned, using the same number of processors, and using the same block size. Other causes of nonreproducibility, however, are based on runtime behavior, such as load balancing or node failures, and thus are inherent in achieving optimal performance at extreme scale.

Randomization

The HPC's future hints at an explosive growth in the volume of data and a relatively dismal growth in the capabilities of communication and IO systems. Under such conditions, it becomes increasingly important to find algorithms that communicate less and perform IO operations even less than that. For an important set of problems in numerical computing, a class of algorithms emerges that seem to be an answer to these challenges: randomization algorithms.

Most of us associate randomized algorithms with Monte Carlo methods and view them as a desperate and final resort because they're highly sensitive to the random-number generator and often produce outputs with low and uncertain accuracy. Recently, new kinds of randomized algorithms have been constructed for a range of problems that play a central role in scientific computing and data analysis, including least-squares problems and matrix approximation problems, such as truncated singular value decomposition and rank-revealing QR decomposition. These new methods are insensitive to the quality of randomness and produce highly accurate results. At the same time, they offer two major advantages, speed and simplicity, as in many cases the randomized algorithm is the fastest option, the simplest, or both.

Randomized methods solve these problems by operating on a sketch of the input matrix instead of the matrix itself. In the case of random sampling, the sketch consists of a small number of carefully selected and rescaled matrix columns or rows (for random projections, the sketch consists of a small number of linear combinations of the matrix columns or rows). In other words, these methods identify a subspace that captures most of the action of the matrix. The matrix is then compressed to this subspace, either explicitly or implicitly, and a deterministic algorithm is applied to the reduced matrix to compute the solution.

Some people in the numerical computing community have embraced the new algorithms. For instance, in their article on the Blendenpick algorithm for the least-squares approximation, Haim Avron, Petar Maymounkov, and Sivan Toledo stated that "randomization is arguably the most exciting and innovative idea to have hit linear algebra in a long time."[23] Others grapple with the probabilistic aspect of the new methods, which, unlike their deterministic counterparts, introduce a nonzero probability of failure. This failure probability, however, is a user-controllable parameter and can be rendered negligible if necessary. The impracticality of demanding constant, complete determinism is captured in a quote from Harold Abelson and Gerald Sussman: "In testing primality of very large numbers chosen at random, the chance of stumbling upon a value that fools the Fermat test is less than the chance that cosmic radiation will cause the computer to make an error in carrying out a 'correct' algorithm."[24]

The new classes of random sampling and projection algorithms offer numerous advantages when dealing with large datasets coming from both scientific (astrophysics, genomics, climate modeling) and commercial applications (social networks, information retrieval systems, financial transactions). In many cases, randomized algorithms beat their classical counterparts in terms of accuracy, speed, and robustness. They utilize modern computer architectures better by exposing higher levels of parallelism than traditional numerical methods. At the same time, they often produce more numerically robust solvers by introducing implicit regularization. While not a silver bullet, a careful application of randomization ideas can lead to powerful frameworks for a range of matrix problems and open the path for using exascale resources to tackle big data challenges.[25,26]

Autotuning

Although Moore's law is still in effect, the multicore revolution has initiated a processor design trend of moving away from architectural features that don't 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 SIMD (single instruction, multiple data) extensions of CPU cores or SIMT (single instruction, multiple thread) 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.

This dramatically affects the way that fast computational kernels are written. Take as an example fast GPU implementations of the famed matrix multiplication or its derivative, the convolution operation in deep learning neural networks. Most loops are tiled, their boundaries fixed, and entire loop nests are completely unrolled into large blocks of straight-line code. The remainders of the iteration space, nondivisible by block sizes, are treated with cleanup codes. Most of the time, vectorization is explicit (for example, SIMD using vector intrinsics, SIMT using CUDA or OpenCL), and so is data motion (loading scratchpad memories, prefetching). This style of code allows for pipelining of floating-point instructions, hiding the latency of load and store instructions, minimizing integer and address arithmetic, and minimizing branching. At the same time, a kernel optimized to that extent for one device is no longer optimal for another. One solution is to hire ninja programmers who recode or retune each kernel from one device to another. Another is to write kernels where tiling sizes and other parameters are tunable and then apply the process of automated software tuning. It's worth mentioning that today's devices have hardware switches that can be controlled in software—for example, Nvidia GPUs have software-controllable L1/shared memory size and software-controllable width of shared memory banks. It's only natural to discover the best settings in the process, otherwise known as *autotuning*.

Automated software tuning was pioneered in projects such as Atlas[27] and Spiral,[28] is the objective of numerous academic projects, and is also practiced by hardware vendors providing libraries such as BLAS for their devices. The basic premise is to explore a search space and find the best performers. The search space can be defined by a set of tunable parameters, code transformations, implementation variants, hardware switches, and so on, and can then be pruned by applying a set of constraints that eliminate obvious underperformers. Finally, it can be searched to find the winners. Exhaustive search, steepest descent methods, and genetic algorithms are all valid approaches.

Another issue is application-level tuning, in which the objective is to maximize performance in a parallel run, executed by a large number of devices (processors or accelerators). The first problem is granularity—the fact that, in most cases, the level of parallelism can be increased at the cost of a drop in serial performance of each device in the mix and at the cost of a hike in communication. The second problem is the tradeoff between load balance and communication, that is, the fact that the former can be improved at the expense of the latter. Unfortunately, making repeated runs to find the optimum is a much less attractive option due to the resources required at large scale. Right now, the solution seems to lie in modeling and simulation.

Moving toward the exascale challenge will require rethinking the entire HPC software stack, as the size, complexity, and heterogeneity of new machines render the existing software infrastructure obsolete. To the rescue come new algorithmic techniques, such as CA algorithms, mixed-precision algorithms, and randomization methods, as well as new programming paradigms, such as dataflow task scheduling, and performance engineering techniques, such as automatic software tuning. Many of these techniques will meet barriers to adoption, as they often venture into unexplored territory of computing, but they're necessary to push the envelope. ■

Acknowledgments

We thank the US Department of Energy, Exascale Computing Project (ECP); the National Science Foundation; and the Russian Federation for sponsoring this work.

References

1. G.E. Moore, "Cramming More Components onto Integrated Circuits," *Electronics*, vol. 38, no. 8, 1965, pp. 114–117.
2. G.E. Moore, "Progress in Digital Integrated Electronics," *IEEE Int'l Electronic Devices Meeting*, 1975, pp. 11–13.
3. R.H. Dennard et al., "Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions," *IEEE J. Solid-State Circuits*, vol. 5, 1974, pp. 256–268.
4. J.G. Koomey et al., "Implications of Historical Trends in the Electrical Efficiency of Computing," *IEEE Annals of the History of Computing*, vol. 33, no. 3, 2010, pp. 46–54.
5. S. Anthony, "Beyond Silicon: IBM Unveils World's First 7nm Chip," *Ars Technica*, 9 July 2015; https://arstechnica.com/gadgets/2015/07/ibm-unveils-industrys-first-7nm-chip-moving-beyond-silicon.
6. S. Anthony, "Intel Forges Ahead to 10nm, Will Move Away from Silicon at 7nm," *Ars Technica*, 23 Feb. 2015; https://arstechnica.com/gadgets/2015/02/intel-forges-ahead-to-10nm-will-move-away-from-silicon-at-7nm/.
7. M. Bohr et al., "Moore's Law Challenges below 10nm: Technology, Design and Economic Implications," *Int'l Solid-State Circuits Conf.*, 2016, p. 1.
8. A. Danowitz et al., "CPU DB: Recording Microprocessor History," *ACM Queue*, vol. 10, no. 4, 2012, pp. 55–63.
9. M. Baboulin et al., "Dense Symmetric Indefinite Factorization on GPU Accelerated Architectures," *Proc. 11th Int'l Conf. Parallel Processing and Applied Mathematics*, 2015, pp. 86–95.
10. D.S. Parker, *Random Butterfly Transformations with Applications in Computational Linear Algebra*, tech.

report CSD-950023, Computer Science Dept., Univ. California, Los Angeles, 1995.

11. M. Baboulin et al., "Accelerating Linear System Solutions Using Randomization Techniques," *ACM Trans. Mathematical Software*, vol. 39, no. 2, 2013, pp. 8:1–8:13.
12. R. Barrett et al., "Algorithmic Bombardment for the Iterative Solution of Linear Systems: A Poly-iterative Approach," *J. Computational and Applied Mathematics*, vol. 74, nos. 1–2, 1996, pp. 91–109.
13. A. Duran et al., "OMPSS: A Proposal for Programming Heterogeneous Multi-core Architectures," *Parallel Processing Letters*, vol. 21, no. 2, 2011, pp. 173–193.
14. "OpenMP Application Program Interface," v. 4.0, OpenMP Architecture Rev. Board, July 2013.
15. G. Ballard et al., "Minimizing Communication in Numerical Linear Algebra," *SIAM J. Matrix Analysis and Applications*, vol. 32, no. 3, 2011, pp. 866–901.
16. M. Mohiyuddin et al., "Minimizing Communication in Sparse Matrix Solvers," *Proc. Conf. High Performance Computing Networking, Storage and Analysis*, 2009, p. 36.
17. J. Demmel et al., "Communication-Optimal Parallel and Sequential QR and LU Factorizations," *SIAM J. Scientific Computing*, vol. 34, no. 1, 2012, pp. A209–A239.
18. G. Ballard et al., "Communication Lower Bounds and Optimal Algorithms for Numerical Linear Algebra," *Acta Numerica*, vol. 23, 2014, pp. 1–155.
19. M. Baboulin et al., "Accelerating Scientific Computations with Mixed Precision Algorithms," *Computer Physics Comm.*, vol. 180, no. 12, 2009, pp. 2526–2533.
20. X.S. Li et al., "Design, Implementation and Testing of Extended and Mixed Precision BLAS," *ACM Trans. Mathematical Software*, vol. 28, no. 2, 2002, pp. 152–205.
21. K. Diethelm, "The Limits of Reproducibility in Numerical Simulation," *Computing in Science & Eng.*, vol. 14, no. 1, 2012, pp. 64–72.
22. G. Bosilca et al., "Algorithm-Based Fault Tolerance Applied to High Performance Computing," *J. Parallel and Distributed Computing*, vol. 69, no. 4, 2009, pp. 410–416.
23. H. Avron, P. Maymounkov, and S. Toledo, "Blendenpik: Supercharging LAPACK's Least Squares Solver," *SIAM J. Scientific Computing*, vol. 32, no. 3, pp. 1217–1236.
24. H. Abelson and G.J. Sussman, *Structure and Interpretation of Computer Programs*, 2nd ed., MIT Press, 1996.
25. N. Halko, P.-G. Martinsson, and J.A. Tropp, "Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions," *SIAM Rev.*, vol. 53, no. 2, 2011, pp. 217–288.
26. M.W. Mahoney, "Randomized Algorithms for Matrices and Data," *Foundations and Trends in Machine Learning*, vol. 3, no. 3, 2011, pp. 123–224.
27. R.C. Whaley, A. Petitet, and J.J. Dongarra, "Automated Empirical Optimizations of Software and the ATLAS Project," *Parallel Computing*, vol. 27, no. 1, 2001, pp. 3–35.
28. M.P. Schel et al., "SPIRAL: A Generator for Platform-Adapted Libraries of Signal Processing Algorithms," *Int'l J. High Performance Computing Applications*, vol. 18, no. 1, 2004, pp. 21–45.

Jack Dongarra holds appointments at the University of Tennessee Knoxville, Oak Ridge National Laboratory, and the University of Manchester. He specializes in numerical algorithms in linear algebra, parallel computing, use of advanced computer architectures, programming methodology, and tools for parallel computers. Dongarra was awarded the IEEE Sid Fernbach Award in 2004, received the first IEEE Medal of Excellence in Scalable Computing in 2008, the first SIAM Special Interest Group on Supercomputing's award for Career Achievement in 2010, and the IEEE IPDPS 2011 Charles Babbage Award. He's a Fellow of the AAAS, ACM, IEEE, and SIAM, and a member of the National Academy of Engineering. Contact him at dongarra@icl.utk.edu.

Stanimire Tomov is a research director and adjunct assistant professor at the University of Tennessee Knoxville. His research interests are in parallel algorithms, numerical analysis, and HPC. Tomov received a PhD in mathematics from Texas A&M University. Contact him at tomov@icl.utk.edu.

Piotr Luszczek is a research director at the University of Tennessee Knoxville's Innovative Computing Laboratory. His core research is centered on performance modeling and evaluation. Contact him at luszczek@icl.utk.edu.

Jakub Kurzak is a research director in the Innovative Computing Laboratory, Department of Electrical Engineering and Computer Science, University of Tennessee Knoxville. His research interests are in HPC with multicore and accelerators. Kurzak received a PhD in computer science from the University of Houston. Contact him at Kurzak@icl.utk.edu.

Mark Gates is a research scientist in the Innovative Computing Laboratory at the University of Tennessee Knoxville, where his interests lie in algorithms for linear algebra on multicore and accelerator-based computers. Gates received a PhD in computer science from the University of Illinois at Urbana-Champaign. Contact him at mgates3@icl.utk.edu.

Ichitaro Yamazaki is a research scientist in the Innovative Computing Laboratory at the University of Tennessee Knoxville, where his interests lie in HPC, especially for linear algebra and scientific computing. Yamazaki received a PhD in computer science from the University of California, Davis. Contact him at iyamazak@icl.utk.edu.

Hartwig Anzt is a research scientist in the Innovative Computing Laboratory at the University of Tennessee Knoxville. His research interests include simulation algorithms, sparse linear algebra, hardware-optimized numerics for GPU-accelerated platforms, and power-aware computing. Anzt received a PhD in mathematics from the Karlsruhe Institute of Technology. Contact him at hanzt@icl.utk.edu.

Azzam Haidar is a research scientist in the Innovative Computing Laboratory at the University of Tennessee Knoxville. His research interests focus on the development and implementation of parallel linear algebra routines for scalable distributed multicore and heterogeneous architectures for large-scale dense and sparse problems. Haidar received a PhD from CERFACS, France. Contact him at haidar@icl.utk.edu.

Ahmad Abdelfattah is a postdoctoral research associate in the Innovative Computing Laboratory at the University of Tennessee Knoxville. His research interests include high-performance numerical linear algebra on GPUs and emerging architectures, including both dense and sparse problems. Abdelfattah received a PhD in computer science from King Abdullah University of Science and Technology. Contact him at ahmad@icl.utk.edu.