PHILOSOPHICAL TRANSACTIONS A

royalsocietypublishing.org/journal/rsta

Research

Cite this article: Dongarra J, Grigori L, Higham NJ. 2020 Numerical algorithms for high-performance computational science. *Phil. Trans. R. Soc. A* **378**: 20190066. http://dx.doi.org/10.1098/rsta.2019.0066

Accepted: 9 December 2019

One contribution of 15 to a discussion meeting issue 'Numerical algorithms for high-performance computational science'.

Subject Areas:
applied mathematics, computational mathematics, computer modelling and simulation

Keywords:
numerical algorithms, numerical linear algebra, rounding errors, floating-point arithmetic, high-performance computing, exascale computer

Author for correspondence:
Nicholas J. Higham
e-mail: nick.higham@manchester.ac.uk

THE ROYAL SOCIETY PUBLISHING

Numerical algorithms for high-performance computational science

Jack Dongarra[1,2,3], Laura Grigori[4] and Nicholas J. Higham[3]

[1]Innovative Computing Laboratory (ICL), University of Tennessee, Knoxville, TN, USA
[2]Oak Ridge National Laboratory, Oak Ridge, TN, USA
[3]Department of Mathematics, University of Manchester, Manchester M13 9PL, UK
[4]Alpines, Inria Paris, Sorbonne Université, Université de Paris, CNRS, Laboratoire Jacques-Louis Lions, 75012 Paris, France

NJH, 0000-0001-5956-4976

A number of features of today's high-performance computers make it challenging to exploit these machines fully for computational science. These include increasing core counts but stagnant clock frequencies; the high cost of data movement; use of accelerators (GPUs, FPGAs, coprocessors), making architectures increasingly heterogeneous; and multiple precisions of floating-point arithmetic, including half-precision. Moreover, as well as maximizing speed and accuracy, minimizing energy consumption is an important criterion. New generations of algorithms are needed to tackle these challenges. We discuss some approaches that we can take to develop numerical algorithms for high-performance computational science, with a view to exploiting the next generation of supercomputers.

This article is part of a discussion meeting issue 'Numerical algorithms for high-performance computational science'.

1. Introduction

High-performance computing (HPC) illustrates well the rapid pace of technological change. A current high-end smartphone can perform linear algebra computations at speeds substantially exceeding that of a Cray-1, which was first installed in 1976 and was widely regarded as the first successful supercomputer.

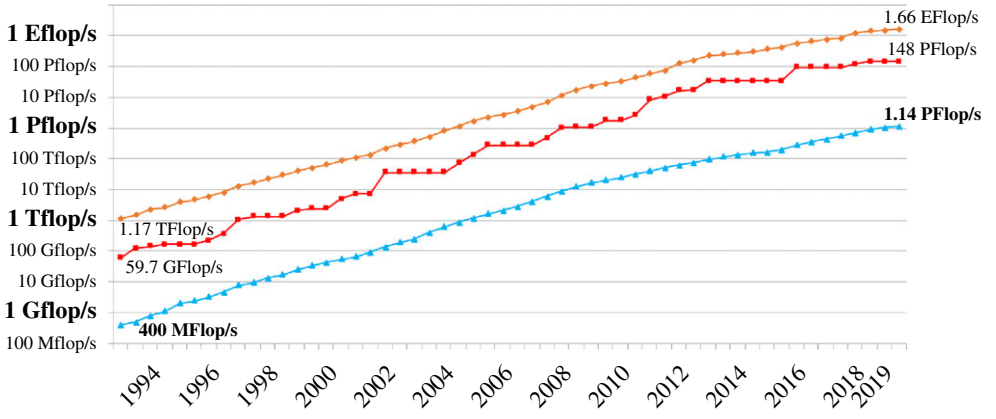

Figure 1. Performance development of supercomputers as tracked by the TOP500. The red line (middle) shows the performance for the highest-performing system on the list, the blue line (bottom) shows the lowest-performing system (number 500) and the orange line (top) shows the sum of the performance of all the systems on the TOP500. (Online version in colour.)

Just a few years ago, teraFLOP/s (10^{12} floating-point operations/second)[1] and terabytes (10^{12} bytes of secondary storage) defined state-of-the-art HPC. Today, those same values represent a PC with an NVIDIA accelerator and local storage. In 2019, HPC is defined by multiple petaFLOP/s (10^{15} floating-point operations/second) supercomputing systems and cloud data centres with many exabytes of secondary storage.

Figure 1 shows this exponential increase in advanced computing capability based on the high-performance LINPACK benchmark [1] used in the TOP500 list of the world's fastest computers [2]. Although the solution of dense linear systems of equations is no longer the best measure of delivered performance on complex scientific and engineering applications, these historical data illustrate how rapidly HPC has evolved. While HPC has benefited from the same semiconductor advances as commodity computing, sustained system performance has risen even more rapidly, due in part to the increasing size and parallelism of high-end systems.

Today, HPC is at a critical juncture, in which several aspects of computer architectures have come together to both create challenges and also offer opportunities.

- Core counts on processors are increasing, but clock frequencies are not (Moore's Law will come to an end in the next few years [3,4]).
- The cost of data movement is starting to dominate the cost of floating-point arithmetic.
- Accelerators (GPUs, FPGAs, coprocessors) are becoming more powerful and more usable, so that heterogeneous architectures are increasingly prevalent.
- Minimizing energy consumption is an increasingly important criterion.
- Low precision floating-point arithmetic is now available in hardware on accelerators and offers greater throughput, albeit less accuracy.

We need a new generation of numerical algorithms that takes account of all these aspects in order to meet the demands of applications on the evolving hardware. Computers attaining an exascale rate of computation (10^{18} floating-point operations per second) will soon be available, and for their success we will need numerical software that extracts good performance from these massively parallel machines. Algorithms, software and hardware are all crucial. Indeed, as noted in [5], 'it is widely recognized that, historically, numerical algorithms and libraries have contributed as much to increases in computational simulation capability as have improvements in hardware'.

[1]Floating-point operation rates are for 64-bit floating-point operations.

In this paper we discuss some of the approaches we can take to developing numerical algorithms for high-performance computational science, and we also look towards the next generation of supercomputers and discuss the challenges they will bring. We begin, in the next section, by looking at how to exploit the availability of different precisions of floating-point arithmetic. In §3 we consider the cost of communicating data and outline some approaches to reducing communication costs. In §4 we consider how to exploit data sparsity, which allows data to be compressed without significant loss of information. Finally, in §5 we discuss exascale computers and some of the algorithmic techniques that may need to be used in software for them.

2. Mixed precision algorithms

For many years, scientific computing has been carried out in single precision (fp32) or double precision (fp64) arithmetic, on hardware supporting the 1985 IEEE arithmetic standard [6]. In 2008, the revised IEEE standard [7] introduced half-precision (fp16) and quadruple precision (fp128) formats. Half-precision was defined for storage only, but several manufacturers now support it for computation. Quadruple precision is available only in software, with the exception of the IBM z13 mainframe systems, designed for business analytics workloads [8]. Another form of half-precision called bfloat16 was introduced by Google on its Tensor Processing Unit and will be supported by Intel in its forthcoming Nervana Neural Network Processor and Cooper Lake processor and on the Armv8-A architecture [9–13]. Table 1 gives the key parameters for all these arithmetics.

We would expect the cost of a floating-point operation to be approximately proportional to the number of bits in the operands, and therefore going from double precision to single precision or single precision to half-precision should give a factor of 2 speed-up in arithmetic costs and a reduction in energy costs. (A notable exception to this rule of thumb was the Sony/Toshiba/IBM (STI) CELL processor, on which single-precision arithmetic was up to 14 times faster than double precision arithmetic [14]). Lower precision data also reduce storage requirements and data movement costs. In fact, even greater benefits are available. NVIDIA's Volta and Turing architectures contain tensor cores that can carry out a block fused multiply-add (FMA) operation $X = AB + C$, where A (fp16), B (fp16) and C (fp16 or fp32) are 4×4 matrices, at fp32 precision, and provide the result X as an fp16 or fp32 matrix; see figure 2. The tensor cores have a throughput of one block FMA per clock cycle and enable a peak fp16 performance eight times faster than for fp32. The tensor cores therefore give both speed and accuracy advantages over pure fp16 computations.

Clearly, then, the use of precisions lower than the traditional single and double is attractive for numerical computing as regards computational cost. Nowhere is this being exploited more than in machine learning, where it is now commonplace to carry out parts of the computations in low precision, possibly in precisions even lower than half-precision [15,16].

However, several issues must be faced in order to make successful use of low precision floating-point arithmetic. First, the IEEE standard fp16 arithmetic has a very narrow range, as shown in table 1: the largest floating-point number is 65504 and the smallest normalized positive number is of order 10^{-4}. Overflow is therefore likely except for 'well behaved' problems and underflow, or the appearance of subnormal numbers (which, since they have leading zeros in the significant, have less precision than normalized numbers) can readily be generated.

More fundamentally, low precision computations will give results of (at best) correspondingly low accuracy, so the question is how such computations can be exploited within an algorithm aiming for higher accuracy. One obvious answer is within a fixed point iteration $x_{k+1} = g(x_k)$, where $g : \mathbb{R}^n \to \mathbb{R}^n$. An example is Newton's method for $f(x) = 0$: $x_{k+1} = x_k - J_f(x_k)^{-1} f(x_k)$, where J_f is the Jacobian of f. The early iterations can be carried out in low precision and the precision gradually increased until the working precision is in use. Assuming that the starting point x_0 and all subsequent exact iterates lie within the region of convergence of an attractive fixed point, the iteration should still converge to that fixed point to the working precision. The speed-up obtained

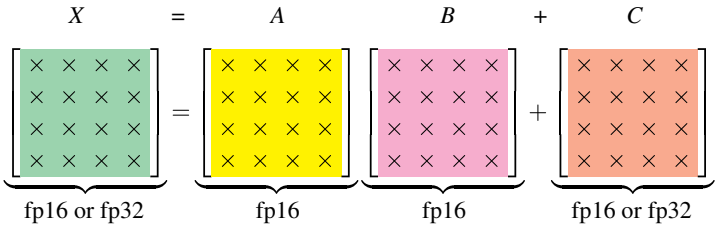

Figure 2. The tensor cores in the NVIDIA Volta and Turing architectures carry out this (possibly) mixed precision 4×4 matrix multiplication and addition in one clock cycle, accumulating the scalar sums at fp32 precision. (Online version in colour.)

Table 1. Parameters for floating-point arithmetics: number of bits in significand (mantissa), including the implicit most significant bit; number of bits in exponent; and, to three significant figures, unit roundoff u, smallest positive (subnormal) number $x_{\min}^{(s)}$, smallest normalized positive number $x_{\min}$ and largest finite number $x_{\max}$. In Intel's bfloat16 specification, subnormal numbers are not supported [10], so any number less than $x_{\min}$ in magnitude is flushed to zero; the value shown holds if subnormal numbers are supported.

	signif.	exp.	u	$x_{\min}^{(s)}$	$x_{\min}$	$x_{\max}$
bfloat16	8	8	3.91×10^{-3}	9.18×10^{-41}	1.18×10^{-38}	3.39×10^{38}
fp16	11	5	4.88×10^{-4}	5.96×10^{-8}	6.10×10^{-5}	6.55×10^{4}
fp32	24	8	5.96×10^{-8}	1.40×10^{-45}	1.18×10^{-38}	3.40×10^{38}
fp64	53	11	1.11×10^{-16}	4.94×10^{-324}	2.22×10^{-308}	1.80×10^{308}
fp128	113	15	9.63×10^{-35}	6.48×10^{-4966}	3.36×10^{-4932}	1.19×10^{4932}

by executing the first few iterations at low precision will clearly be limited in general. However, if the iteration has large set-up costs, greater efficiencies are possible, as we now explain.

Consider a linear system $Ax = b$, where $A \in \mathbb{R}^{n\times n}$ is non-singular. The standard method of solution is to compute an LU factorization $A = LU$ and then solve the two triangular systems $Ly = b$ and $Ux = y$. In practice, pivoting is used in the LU factorization: either partial pivoting (row interchanges) if A is dense or pivoting for stability and to preserve sparsity (row, and possibly column, interchanges) if A is sparse; for simplicity we omit these interchanges from the equations. The key to exploiting different precisions is to observe that most of the work in solving $Ax = b$ lies in computing the LU factorization: for dense matrices this costs $O(n^3)$ flops (floating-point operations) compared with the $O(n^2)$ flops required to solve a triangular system. We can compute an LU factorization $A \approx \hat{L}\hat{U}$ in low precision and use it to obtain an initial guess x_0. Then we can refine the solution by what amounts to Newton's method for $f(x) = b - Ax$, namely $x_{k+1} = x_k + A^{-1}(b - Ax_k)$. But of course we cannot exactly apply the matrix A^{-1}, so we replace it by $\hat{U}^{-1}\hat{L}^{-1}$ and implement

— compute $r_k = b - Ax_k$ (at the working precision),
— solve $\hat{L}\hat{U}d_k = r_k$ (at low precision),
— update $x_{k+1} = x_k + d_k$ (at the working precision).

We hope that a few iterations of these equations will be enough to yield a backward error $\|b - Ax_k\|/(\|A\|\|x_k\| + \|b\|)$ of order the working precision. Here, $\|\cdot\|$ denotes any standard vector norm and the corresponding subordinate matrix norm.

This process is known as iterative refinement. It is an old method going back to the beginning of the digital computer era, though in its original usage the residual is computed at twice the working precision and all the other steps are done at the working precision. It was first proposed in the form shown here by Langou *et al.* [17], who used single and double precisions.

Convergence will be achieved if $\kappa(A)u_\ell$ is sufficiently less than 1, where $\kappa(A) = \|A\|\|A^{-1}\|$ is the matrix condition number and u_ℓ is the unit roundoff for the low precision arithmetic. This is a severe restriction when we take the low precision to be fp16, as then we need $\kappa(A) \leq 10^4$. Carson & Higham [18,19] show how to greatly widen the class of problems to which iterative refinement is applicable by using the LU factors as preconditioners and employing *three* precisions. Denote the unit roundoff for the working precision by u and the unit roundoff for a possibly higher precision by u_r. Thus $u_\ell > u \geq u_r$. The steps above now become

— compute $r_k = b - Ax_k$ at precision u_r,
— solve $MAd_k = Mr_k$, where $M = \hat{U}^{-1}\hat{L}^{-1}$ by GMRES (at precision u, with M applied at precision u_r),
— update $x_{k+1} = x_k + d_k$ at precision u.

GMRES [20] is an iterative Krylov subspace method that requires matrix–vector products with the coefficient matrix, which here is MA, and these products reduce to a product with A and triangular solve with $\hat{L}$ and $\hat{U}$. Carson & Higham show that this GMRES-IR method based on three-precisions converges under much weaker conditions than the traditional form of iterative refinement. If the three precisions are fp16, fp64 and fp128, then we need $\kappa(A) \leq 10^{16}$ for convergence, and if the algorithm is relaxed so that extra precision is not used in computing r_k and applying M then we need $\kappa(A) \leq 10^7$ [21]. For GMRES-IR to be effective, we need GMRES to converge quickly on the preconditioned system; it usually does, but there is a lack of theoretical guarantees.

Haidar *et al.* [22,23] show that by taking advantage of the tensor cores on an NVIDIA V100 GPU, GMRES-IR can bring a speed-up of 4 over an optimized double precision solver and can provide an energy reduction of 80%. Moreover, GMRES-IR has been shown to perform up to three times faster than an optimized double precision solver at scale on the Summit machine [24], which heads the November 2019 TOP500 list. So with this new twist, the old method of iterative refinement provides a powerful way to exploit low precision arithmetic. The possibility of overflow and underflow threatens to render the method ineffective for real-life problems, because of their possible bad scaling, but Higham *et al.* [25] show how diagonal scaling can be used to greatly reduce the possibility of overflow and underflow.

GMRES-IR can be extended to symmetric positive definite linear systems, with the use of low precision Cholesky factorization [26]. Here, one must handle the possibility of the matrix losing definiteness when rounded to lower precision. It can also be extended to linear least-squares problems [26,27].

The MAGMA library (Matrix Algebra on GPU and Multicore Architectures)[2] [28] supports GMRES-IR, with fp16 or fp32 as the lower precision arithmetic.

GMRES-IR is used in the new HPL-AI benchmark, which 'seeks to highlight the emerging convergence of HPC and artificial intelligence (AI) workloads' [29]. The benchmark uses double precision for u and u_r and a lower precision (expected to be chosen as half-precision) for u_ℓ.

A concern when low precision arithmetic is used is that rounding errors might cause algorithms to lose all their accuracy. The reason is that rounding error bounds for basic linear algebra kernels are typically of order nu or larger, where n is the problem dimension and u is the unit roundoff of the working precision. In fp16, nu exceeds 1 for n as small as 2049. However, rounding error bounds are worst-case bounds that are usually not attainable. In practice, rounding errors often cancel to some extent, and so the worst case is not a predictor of the typical behaviour. Wilkinson [30, p. 318] noted that 'In general, the statistical distribution of the rounding errors will reduce considerably the function of n occurring in the relative errors. We might expect in each case that this function should be replaced by something which is no bigger than its square root.' Higham & Mary [31] give a rigorous probabilistic rounding error analysis that justifies this rule of thumb, showing that under reasonable statistical assumptions

[2]See https://icl.cs.utk.edu/magma/

a number of standard error bounds still hold with high probability if constants are replaced by their square roots. We are not yet seeing rounding errors make numerically stable linear algebra algorithms unusable for low precisions, but more understanding is certainly needed.

In using mixed precision, it is necessary to decide how to choose the precisions in different parts of an algorithm. Rounding error analysis of the algorithm is complicated by the need to track the different precisions and by the possible use of a mixed-precision block FMA. Blanchard *et al.* [32] analyse the use of a mixed-precision block FMA in the numerical linear algebra kernels of matrix multiplication and LU factorization. Their analysis provides insight into finding a good trade-off between performance and accuracy when these computations use the tensor cores on an NVIDIA V100 GPU.

Many authors are experimenting with low precision arithmetic in machine learning and other areas and are finding that it does not cause any deterioration in the quality of the computed results. One example is in climate change [33–35]. Another is work reporting that in a Monte Carlo simulation single-precision arithmetic was replaced by bfloat16 without any loss of accuracy [36]. We also mention that some hardware designed for specific applications employs low precision because of its lower energy consumption and the reduced size of the chips; see, for example, [37,38].

We note, however, the importance of carefully analysing the effects of lowering arithmetic precision. To take a very simple example, suppose we want to evaluate $\|x\|_2$ for $x = [\alpha\ \alpha]^T$ as $(x_1^2 + x_2^2)^{1/2}$ in fp16 arithmetic. With $\alpha = 10^{-4}$ the computed result is 0, and hence the relative error is 1, because α^2 underflows to zero. In practice, one can scale the data to avoid this problem, but this example serves to illustrate that low precision should not be used blindly.

Numerical experiments are essential to gain understanding, but they are not possible if the relevant hardware is not available or if one wishes to try potential new floating-point formats. In this case, simulation is necessary. Higham & Pranesh [39] present a tool for simulating arithmetics of different precisions in Matlab, give pointers to simulations in other languages, and explain why some simulations deliver results that are too accurate.

3. Algorithms minimizing data transfer

The cost of communication, that is the cost of transferring data between different processing units or between different levels of the memory hierarchy, dominates the cost of many algorithms, in terms of both time and energy consumption. Several studies outline a large disproportion between the improvements in processor technologies and improvements in memory or interconnect technologies. Writing in 1995, Wulf & McKee [40] noted that 'we are going to hit a wall in the improvement of system performance unless something basic changes'. We are clearly also facing the interconnect network wall, with the network latency and bandwidth improvements lagging well behind improvements in processor technologies. A comprehensive report studying data from 1995 to 2004 [41] concluded that while the time per flop had improved at a rate of 59% yearly, the network latency had improved at a rate of only 15% yearly, while DRAM latency had improved by only 5.5% yearly. Nowadays, even if processor frequencies tend not to vary by much, per socket flop performance continues to improve by, e.g., increasing the number of floating-point operations per cycle per core or the number of cores per socket. The network latency, as measured at the user level in an MPI call, is of a few microseconds; see e.g. [42] for recent data.

Many works have addressed the communication problem. One of the first attempts to reduce communication in dense linear algebra was by Barron & Swinnerton-Dyer [43] in 1960. They were using the EDSAC 2 computer to solve linear systems of equations by Gaussian elimination, and the largest system that could be stored in the main store had 31 equations. As pointed out in the paper, the subroutine implementing Gaussian elimination was the first library subroutine written for the EDSAC 2. Nowadays, this is still one of the first routines optimized on a new supercomputer, in particular because it is used in the high-performance LINPACK benchmark that determines the TOP500 list [2]. For solving systems with more than 31 equations, the data

had to be read from and written back to auxiliary magnetic-tape storage during the elimination. The authors introduce two algorithms that aim at reducing the data transferred between the main store and the auxiliary magnetic-tape. The first one uses a block LU factorization with partial pivoting to ensure numerical stability. This block factorization is the basis of the algorithms implemented in most of the dense linear algebra libraries, going from LAPACK [44] and ScaLAPACK [45] to modern versions based on runtime systems as PLASMA [46]. The second algorithm introduces a different pivoting strategy, a block pairwise pivoting strategy, and it can be shown that with this pivoting strategy the algorithm minimizes communication between two different levels of the memory hierarchy. However, while the factorization remains stable for systems of small size, it is observed in [47] that it might become unstable for systems of the large sizes encountered nowadays.

In recent years, a different approach has been introduced for reducing, or even minimizing, communication in numerical linear algebra, which relies on different ways to compute the same algebraic operations. For example, the LU factorization of a matrix is computed by using a different pivoting strategy than partial pivoting, while the numerical stability is still ensured. Indeed, since the numerical approach used to compute the algebraic operation changes, special care needs to be taken to ensure that the novel algorithms remain at least as stable as conventional ones. These algorithms are referred to as *communication avoiding*.

Some of the major results obtained in this context are asymptotic lower bounds on communication for dense linear algebra operations based on direct methods, such as LU factorization, QR factorization, rank revealing factorizations and singular value or eigenvalue decompositions. These lower bounds indicate the minimum amount of communication, in terms of the number of messages and volume of communication that needs to be transferred when computing such a decomposition on a sequential machine with two levels of memory or between the processors of a parallel machine. The pioneering results obtained by Hong and Kung in 1981 [48] provide lower bounds on the volume of data that has to be transferred during the multiplication of two matrices between two different memories, a memory of size M in which the matrices do not fit entirely, and memory of large size in which the matrices can be stored. These bounds were extended in [49] to the parallel case and proved by using the Loomis and Whitney inequality, which enables one to bound the number of floating-point operations that can be performed given the data available in the memory of size M. They were later extended to bound both the volume of communication and number of messages and to LU factorization and QR factorization under certain assumptions in [50], and then to a wider variety of direct linear algebra algorithms [51].

We explain in more detail the lower bounds on communication for the case in which the linear algebra operation involves dense matrices of dimension $n \times n$ and is executed in parallel on P processors, and the memory available per processor is of the order of n^2/P. Then the lower bounds on the volume of communication (# words) and on the number of messages that need to be exchanged between processors are

$$\#\,\text{words} \geq \Omega\left(\frac{n^2}{\sqrt{P}}\right), \quad \#\,\text{messages} \geq \Omega\left(\sqrt{P}\right). \tag{3.1}$$

Here, $f(n) = \Omega(g(n))$ means that there is a non-zero constant c such that $f(n) \geq cg(n)$ for all sufficiently large n. Cannon's algorithm [52] for dense matrix multiplication and block Cholesky factorization both attain these bounds and hence minimize communication asymptotically in terms of both volume of communication and number of messages. However, most of the remaining conventional algorithms attain the lower bound on the volume of communication but do not attain the lower bound on the number of messages. It is to be noted that the lower bound on the number of messages does not depend on the size of the data, but depends only on the number of processors. And for most conventional algorithms, the number of messages increases proportionally with the dimensions of the matrices. For example, to ensure numerical stability, partial pivoting is used during the LU factorization: at each step it finds the off-diagonal

element of maximum magnitude in the pivot column and permutes the corresponding row to the diagonal position. This leads to a collective communication among all processors owning the corresponding column and hence overall the number of messages is proportional to the number of columns in the input matrix. A different pivoting strategy is used in the communication avoiding LU factorization introduced in [53], referred to as tournament pivoting, which allows the LU factorization of a block of columns to be computed with only $\log_2 P$ messages. The same approach can be used to compute a low-rank approximation of a matrix or its rank revealing factorization, while also ensuring bounds on the errors of the approximation. Perhaps one of the most useful examples is an algorithm referred to as TSQR, which computes the QR factorization of a tall and skinny matrix, that is a matrix with few columns and many rows, with only $\log_2 P$ messages. To the best of our knowledge, the idea goes back to [54,55] with a presentation in a general setting and proofs that it minimizes communication provided in [56]. With some additional computation [57], the output provided by TSQR can be identical to the output provided by a conventional algorithm as implemented in LAPACK. Speed-ups resulting from using these communication avoiding algorithms are reported in [57], for example.

While direct methods in dense linear algebra are well-understood today from a communication complexity point of view, deriving lower bounds on communication for sparse matrices is a much harder problem, for both iterative and direct methods. It is possible to apply directly the bounds derived for dense matrices to direct methods involving matrices with arbitrary sparsity structure, but in general, these bounds can become vacuous. A few results exist, though, for matrices with specific sparsity structures. Tight lower bounds on communication for the sparse Cholesky factorization of a model problem are derived in [58], and the multiplication of Erdős-Rényi matrices is discussed in [59]. Reducing communication in iterative methods is a challenging topic, with an additional difficulty being the fact that the convergence of iterative methods depends on the spectral properties of the matrices involved. Approaches such as s-step methods (e.g. [60,61] and the references therein) or enlarged Krylov methods [62,63] are actively investigated.

Many open questions remain to be addressed beyond linear algebra. In a general setting, the problem to tackle is the following. Given an algorithm based on multiple nested loops that reference arrays with different dimensions, the goal is to identify lower bounds on communication for this algorithm, and subsequently an optimal loop tiling that allows those bounds to be attained. Results along this direction are presented in [64] and they are based on the discrete multilinear Holder–Brascamp–Lieb (HBL) inequalities. The polyhedral model used in the compiler community is another approach for reorganizing nested loops to reduce data movement, and there might be interesting connections between the two approaches. Avoiding communication in machine learning algorithms is a timely topic, with several recent results obtained for, e.g., primal and dual block coordinate descent methods [65], least angle regression [66] and with many open venues to explore.

Dealing with data in high dimensions represented as tensors is another timely and challenging topic. There are few existing results on communication bounds for tensors, e.g. bounds for symmetric tensor contractions are derived in [67], or for the volume of communication of metricized tensor times Khatri–Rao product in [68]. One interesting question is to understand if algorithms such as alternating least squares and density matrix renormalization group are able to minimize communication.

4. Exploiting data sparsity

In this section, we address the problem of dealing with large volumes of data by exploiting 'data sparsity'. This problem arises in many applications, ranging from scientific computing where complex phenomena are simulated over large domains and long periods of time, to machine learning where large volumes of data are processed. Data sparsity refers to the fact that, due to redundancy, the data can be efficiently compressed while controlling the loss of information.

A classic example in this context is the n-body problem in physics for which some forces between all pairs of n particles are computed with $O(n)$ flops by using the fast multipole method (FMM) [69], while a direct evaluation of these forces requires $O(n^2)$ flops. This method relies on treating separately the contributions from near particles, which are computed directly, from contributions from distant particles, which are evaluated by using multipole expansions. This approach can be used in other applications, as for example in the discretization of boundary integral operators. The operator in this case is defined as $G(x_i, y_j)$, where $G : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{C}$ is a kernel operator, with $d \in \mathbb{N}^*$, and the interaction domains are $X := \{x_1, \ldots, x_m\}$ and $Y := \{y_1, \ldots, y_n\}$. The associated dense matrices, referred to as BEM matrices, are full rank, but the kernel evaluation can be again separated into interactions which are computed directly and interactions which are evaluated fast through compression. The original FMM method is kernel-dependent, but later on several kernel-independent methods have been proposed, kernel independent FMM, e.g. [70], hierarchical matrices or H^2 matrices. For a discussion of hierarchical matrices see e.g. [71–74].

An algebraic compression method used by all communities dealing with large volumes of data nowadays is the singular value decomposition (SVD) of a matrix. Given $A \in \mathbb{R}^{m \times n}$, the problem is to compute a rank-k approximation $A_k = ZW^T$, where $Z \in \mathbb{R}^{m \times k}$, $W^T \in \mathbb{R}^{k \times n}$ and $k \ll \min(m, n)$. We suppose for simplicity that $m \geq n$ and that the matrices are real. When the matrix A is sparse, in the sense that there are few elements of A that are non-zero, an additional goal is to obtain factors Z and W that preserve the sparsity of A. Very often the matrix A is used in an iterative procedure, such as a Krylov subspace solver. At each iteration of these algorithms, the multiplication of A with a vector is approximated by the multiplication of A_k with the vector, and this leads to significant computational and memory savings.

The SVD factors a matrix as $A = U\Sigma V^T$, where $U \in \mathbb{R}^{m \times m}$ and $V \in \mathbb{R}^{n \times n}$ are orthogonal matrices whose columns are the left and right singular vectors of A, respectively, and $\Sigma \in \mathbb{R}^{m \times n}$ is a diagonal matrix whose diagonal elements are $\sigma_1(A) \geq \cdots \geq \sigma_n(A) \geq 0$. The rank-$k$ truncated SVD of A is $A_{\mathrm{opt},k} = U_k \Sigma_k V_k^T$, where Σ_k is the leading $k \times k$ principal submatrix of Σ containing on its diagonal the largest singular values $\sigma_1(A), \ldots, \sigma_k(A)$, U_k is formed by the k leading columns of U, and V_k is formed by the k leading columns of V. A result of Eckart & Young [75] shows that the best rank-k approximation of A in both the 2-norm and the Frobenius norm is $A_{\mathrm{opt},k} = U_k \Sigma_k V_k$:

$$\min_{\mathrm{rank}(A_k) \leq k} ||A - A_k||_2 = ||A - A_{\mathrm{opt},k}||_2 = \sigma_{k+1}(A) \tag{4.1}$$

and

$$\min_{\mathrm{rank}(A_k) \leq k} ||A - A_k||_F = ||A - A_{\mathrm{opt},k}||_F = \left(\sum_{j=k+1}^{n} \sigma_j^2(A) \right)^{1/2}. \tag{4.2}$$

However, computing the SVD of a large matrix is prohibitevely expensive, and an active research area focuses on algorithms that approximate the SVD, compromising accuracy for speed. There are three properties of interest for these approximations: we follow here the definitions from [76]. The first concerns the low-rank approximation and compares the error of the approximation, measured in the 2-norm $\|A - A_k\|_2$ or Frobenius norm $\|A - A_k\|_{\mathrm{F}}$, with the optimal error obtained by the SVD from (4.1) and (4.2), respectively. The second one focuses on the approximation of the k leading singular values of A by the singular values of A_k. The approximation is called spectrum preserving if $1 \leq \sigma_i(A)/\sigma_i(A_k) \leq \gamma$ for all $i = 1, \ldots, k$ and some $\gamma \geq 1$, with γ being a low degree polynomial in k and the dimensions of the matrix m, n. The third property focuses on the approximation of the trailing $n - k$ singular values of A. The low-rank approximation is a kernel approximation of A if $1 \leq \sigma_j(A - A_k)/\sigma_{k+j}(A) \leq \gamma$, for all $j = 1, \ldots, n - k$ and some $\gamma \geq 1$.

There are two different classes of algorithms, deterministic algorithms, whose guarantees hold deterministically, and randomized algorithms, whose guarantees hold with high probability. Several different deterministic algorithms exist. The Lanczos method (see, e.g. [77]) is a computationally efficient iterative algorithm, in particular for sparse matrices, for approximating

the leading singular values and associated singular vectors of a matrix. Rank revealing factorizations based on QR factorization with column permutations or LU factorization with row and column permutations are also efficient approaches for computing a low-rank approximation. The first factorization of this type, the QR factorization with column pivoting, referred to as QRCP, was introduced in [78]. It can be shown that with this algorithm, $\|A - A_k\|_2 \leq 2^k \sqrt{n-k}\,\sigma_{k+1}(A)$, but in practice QRCP works very well. The strong rank revealing QR factorization [79] or LU factorization with column and row tournament pivoting [80] provide guarantees for the low-rank approximation, spectrum preserving, and kernel approximation properties, with γ being a low degree polynomial in m, n and k. These factorizations can be computed in $O(mnk)$ flops. In the case of strong rank revealing factorization, $\gamma = \sqrt{1 + f^2 k(n-k)}$, where f is a small constant. In the case of LU with column and row tournament pivoting, $\gamma = (1 + F_{\mathrm{TP}}^2(n-k))^{1/2}(1 + F_{\mathrm{TP}}^2(m-k))^{1/2}$, where $F_{\mathrm{TP}} \leq (2k)^{-1/2}(n/k)^{\log_2(\sqrt{2}fk)}$ for binary tree-based tournament pivoting. The spectrum-revealing property holds for a submatrix of A_k. Since f and k are small, F_{TP} can be seen as a low degree polynomial in n and k. For more details see [80].

Randomized algorithms are based on the technique of linear sketching. The so-called randomized SVD determines a linear sketching of $A \in \mathbb{R}^{m\times n}$ by multiplying A with a random matrix $\Omega_1 \in \mathbb{R}^{n\times \ell}$ from a certain distribution on matrices; it then computes an orthonormal basis Q_1 for $A\Omega_1$ and finally obtains the low-rank approximation $A_k = Q_1 Q_1^T A$. Here, $\ell \geq k$ is an oversampling parameter. For an overview, see e.g. [81,82]. The random matrix Ω_1 is chosen to be a Johnson–Lindenstrauss transform or a fast Johnson–Lindenstrauss transform, as in the sub-sampled randomized Hadamard transform (SRHT) introduced in [83]. The computation of $A\Omega_1$ costs $2mn\log_2 \ell + 1$ flops when Ω_1 is an SRHT ensemble and A is dense, and the QR factorization of $A\Omega_1$ costs $2m\ell^2$ flops. However computing $Q_1^T A$ still costs $2mn\ell$ flops, which is comparable to the cost of deterministic algorithms based on QR and LU rank revealing factorizations. Hence randomized LU-like approaches have been introduced later to decrease this cost, for example, the randomized SVD with row selection algorithm [81]. For a study of the connection between deterministic and randomized low-rank approximation algorithms, see [76].

In terms of guarantees, the literature focused on bounding the error of the approximation, typically in the Frobenius norm, $\|A - A_k\|_{\mathrm{F}}$, by using the Johnson–Lindenstrauss embedding and oblivious subspace embedding properties of these ensembles. The spectrum preserving and kernel approximation properties of randomized algorithms were studied only very recently in [76]. As a main result, [76] introduces sharp bounds of a randomized generalized LU factorization that uses SRHT ensembles to sketch both the columns and the rows of A. The resulting low-rank approximation is computed in $O(nm\log_2(\ell') + m\ell\ell')$ flops, where $\ell' > \ell > k$ represent the oversampling of rows and columns, respectively, and are a polylog-factor larger than m, n, k and the probability δ.

Communication plays an important role in the efficiency of an algorithm, as discussed in §3. When $k \ll \min(m,n)$ and the matrix A is distributed over many processors, a lower bound on the number of messages that need to be exchanged between processors to compute a rank-k approximation is $\Omega(\log_2 P)$. Deterministic algorithms such as QR with column pivoting [78] or strong rank revealing QR [79] are not able to attain this bound, as they require exchanging $\Omega(k\log_2 P)$ messages between processors. Communication avoiding algorithms, described in §3, solve this problem by using, for example, tournament pivoting techniques to select columns and/or rows in the context of rank revealing LU and QR factorizations.

The choice of the most appropriate low-rank approximation algorithm depends on the problem that needs to be solved. There are situations in which the low-rank approximation needs to be computed while traversing only once (or $O(1)$ times) the rows or columns of A, as in stream algorithms. Another common situation is when the matrix A is given only implicitly, for example, as a product of several matrices. In this case, the Lanczos method or randomized algorithms are suitable since they require only multiplying a vector by A. By contrast, factorization-based methods require forming the matrix A explicitly. In other cases, as for example, in boundary element methods (BEM), computing all the elements of the BEM matrix is too expensive with

respect to the overall cost of multiplying it with a vector using either FMM or hierarchical matrix representation. In this case, low-rank sub-blocks of BEM matrices are compressed by using the so-called adaptive cross approximation (ACA) method (see, e.g. [71]). The important problem is to compute a low rank approximation while preserving interpretability of the original data. One such example is the feature extraction problem in data mining. When A is a non-negative matrix (all its elements are non-negative), the NMF factorization computes a low-rank approximation ZW^T such that both Z and W are non-negative matrices.

A challenging topic that we do not discuss here is compressing data in high dimensions that is represented as a tensor, which is a multidimensional array. For a review on this topic, see e.g. [84–86]. This problem arises in a large variety of applications, such as high-dimensional parametric PDEs, electronic structure calculations, and machine learning. Tensors do not enjoy the optimality properties of the rank-k truncated SVD for matrix compression, as in (4.1) and (4.2). The problem of finding a best low-rank approximation by factoring a tensor into a sum of component rank-one tensors (by using the CP decomposition) is ill posed [87] for many ranks. For example, an approximation of a lower rank can provide better errors. As this illustrates, there are many fascinating open problems, including deriving (parallel) algorithms for computing low-rank tensor approximations and dealing with tensors that have high rank but are formed from many subtensors of small rank.

Algorithms that exploit data sparsity should not, of course, sacrifice numerical stability, which is especially important if they are implemented in precisions lower than double. Positive results in this regard are given in [88,89].

5. Towards HPC's next scale

HPC combines hardware, software and algorithms to deliver the highest capability at a given time. In the 1980s, vector supercomputing dominated HPC, notably in the systems designed by the late Seymour Cray. The 1990s saw the rise of massively parallel processing and shared-memory multiprocessors, built by Silicon Graphics, Thinking Machines and others. In turn, clusters of commodity (Intel/AMD x86) and purpose-built processors (e.g. IBM's BlueGene), dominated the 2000s. Today, those clusters have been augmented with computational accelerators and GPUs and HPC means computing in the range of hundreds of petaFLOP/s. We are on the verge of reaching the next big milestone in HPC: the exascale computing era.

Planning for exascale computers has been underway in the USA for around a decade [90–92]. The research and development costs to create an exascale computing system have been estimated to exceed $1 billion, with annual operating costs of tens of millions of dollars, so these systems require significant investment from governments and research agencies. The European Union, Japan and China all have next-generation computing system research and development projects in competition with the USA [93–95].

We briefly discuss some challenges that will arise in using exascale computers, to add to the mixed precision, data movement and data sparsity challenges described above.

(a) Asynchronous algorithms

To solve very large problems by iteration on parallel architectures, we can reduce communication and synchronization overheads by removing the requirement that an iterate is fully updated before moving on to the next iteration. A processor is allowed to work with whatever data it has without waiting for new data to arrive from other processors. Thus the order in which components of the solution are updated is arbitrary and the past values of components are used in the updates. This is an old idea [96] that has attracted renewed interest in recent years; see e.g. [97,98]. The hope is that reducing communication and synchronization will result in shorter time to the solution even though more iterations may be required. In the future, iterative algorithms that are (almost) totally asynchronous may become competitive for a wide range of application problems.

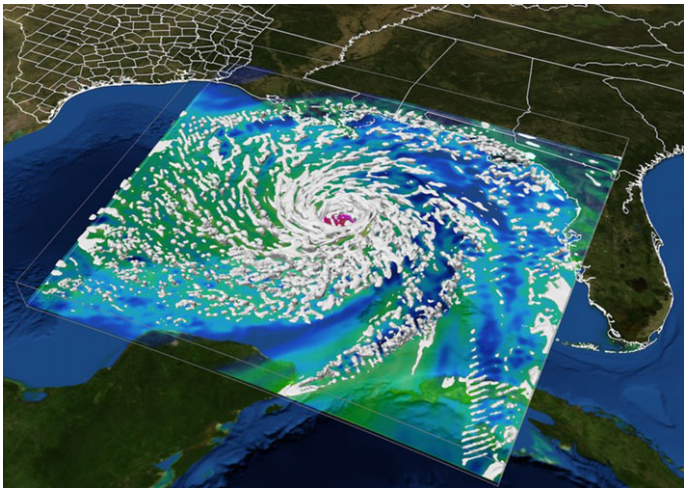

Figure 3. A computer visualization, created by Texas Advanced Computing Center (TACC) supercomputer Ranger, of Hurricane Ike. It shows the storm developing in the Gulf of Mexico before making landfall at the Texas coast. (Online version in colour.)

(b) Autotuning

Given the diverse, evolving and possibly heterogeneous architectures on which software must run, automatic ways to select the various algorithmic parameters will be increasingly needed in order to achieve good performance, energy efficiency, load balancing and so on. Autotuning is already routinely used for core numerical linear algebra algorithms, see, e.g. [99,100], and references therein.

(c) Fault tolerance

Exascale systems will have such a large number of processors that failure of one or more processors during a run will be routine, so the whole software system must be able to cope with such failures. Restarting a computation, possibly from a checkpoint, is not sufficient. Algorithm-based fault-tolerance techniques already exist, e.g. [101], but new paradigms will be needed for handling faults at both the application level and the system level.

(d) Randomized algorithms

Randomization is an increasingly popular technique that can take several forms. Randomly transforming the data (without changing the solution) can help avoid the need for costly operations such as pivoting in a factorization [102], while, as discussed in the previous section, random sampling of a matrix can approximate a subspace with computations dominated by matrix multiplication [81,103].

(e) Exploiting artificial intelligence

AI, including machine learning and deep learning, offers the promise of being able to boost HPC applications to produce superior capabilities and performance. This is an increasingly active and rapidly advancing research field. At SC18 (the International Conference for HPC, Networking, Storage, and Analysis), AI featured heavily in the applications nominated for the prestigious Gordon Bell award of the Association for Computing Machinery [104], including in work identifying extreme weather patterns from high-resolution climate simulations, identifying materials' atomic-level information from electron microscopy data and simulating earthquake

physics in urban environments. Future research must determine what applications can benefit from AI and how it can best be implemented at large scale.

To mention just one benefit that exascale systems will bring, weather models will be able to predict the timing and path of severe weather events, such as hurricanes, more rapidly and with more accuracy by using much higher spatial resolution, incorporating more physics and assimilating more observational data [105]. Figure 3 shows an existing visualization.

Meeting the hardware and software challenges posed by HPC will produce considerable trickle-down benefits. These include enhancements to smaller computer systems and many types of consumer electronics—from smartphones to cameras—as available devices become smaller, faster, more fault tolerant and more energy efficient. The benefits of exascale computing will flow not just from classical simulations but also from large-scale data analysis, machine learning and deep learning, and often the integration of all three approaches.

Data accessibility. This article has no additional data.

Authors' contributions. All authors drafted and revised the manuscript. All authors read and approved the manuscript.

Competing interests. We declare we have no competing interests.

Funding. The work of J.D. was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the US Department of Energy's Office of Science and National Nuclear Security Administration. The work of L.G. has received funding from the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme (grant agreement no. 810367). The work of N.J.H. was supported by the Royal Society.

Acknowledgements. We thank Massimilano Fasi, Theo Mary, Mantas Mikaitis, Srikara Praensh and Mawussi Zounon for their comments on a draft manuscript.

References

1. Dongarra JJ. 1988 The LINPACK benchmark: an explanation. In *Supercomputing, First International Conference, Athens, Proceedings* (eds EN Houstis, TS Papatheodorou, CD Polychronopoulos). Lecture Notes in Computer Science, vol. 297, pp. 456–474. Berlin, Germany: Springer.
2. Meuer H, Strohmaier E, Dongarra J, Simon H, Meuer M. TOP500 Supercomputer Sites. http://www.top500.org.
3. Shalf JM, Leland R. 2015 Computing beyond Moore's law. *Computer* **48**, 14–23. (doi:10.1109/MC.2015.374)
4. Shalf J. 2020 The future of computing beyond Moore's law. *Phil. Trans. R. Soc. A* **378**, 20190061. (doi:10.1098/rsta.2019.0061)
5. Group EMW. 2004 Applied mathematics research for exascale computing. Report US Department of Energy, Office of Science Advanced Scientific Computing Research Program.
6. IEEE Computer Society 1985 IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985. New York, NY: Institute of Electrical and Electronics Engineers.
7. IEEE Computer Society 2008 IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2008 (revision of IEEE Std 754-1985). New York, NY: IEEE Computer Society.
8. Lichtenau C, Carlough S, Mueller SM. 2016 Quad precision floating point on the IBM z13. In *2016 IEEE 23nd Symp. Computer Arithmetic (ARITH), Santa Clara, CA*, pp. 87–94.
9. Feldman M. 2018 Intel Lays Out Roadmap for Next Three Xeon Products. See https://www.top500.org/news/intel-lays-out-roadmap-for-next-three-xeon-products/ (accessed 5 June 2019).
10. Intel Corporation. 2018 BFLOAT16—Hardware Numerics Definition. White paper. Document number 338302-001US.
11. Rao N. 2018 Beyond the CPU or GPU: Why Enterprise-Scale Artificial Intelligence Requires a More Holistic Approach. See https://newsroom.intel.com/editorials/artificial-intelligence-requires-holistic-approach (accessed 5 November 2018).
12. Lutz DR. 2019 ARM Floating Point 2019: Latency, Area, Power. In *2019 IEEE 26th Symp. on Computer Arithmetic (ARITH), Kyoto, Japan*, pp. 97–98. Piscataway, NJ: IEEE.

13. Stephens N. 2019 BFloat16 processing for Neural Networks on Armv8-A. See https://community.arm.com/developer/ip-products/processors/b/ml-ip-blog/posts/bfloat16-processing-for-neural-networks-on-armv8_2d00_a (accessed 14 October 2019).
14. Kurzak J, Dongarra J. 2007 Implementation of mixed precision in solving systems of linear equations on the cell processor. *Concurrency Computat. Pract. Exper.* **19**, 1371–1385. (doi:10.1002/cpe.1164)
15. Gupta S, Agrawal A, Gopalakrishnan K, Narayanan P. 2015 Deep learning with limited numerical precision. In *Proc. 32nd Int. Conf. on Machine Learning, JMLR: Workshop and Conference Proceedings, Lille, France*, vol. 37, pp. 1737–1746.
16. Svyatkovskiy A, Kates-Harbeck J, Tang W. 2017 Training distributed deep recurrent neural networks with mixed precision on GPU clusters. In *MLHPC'17: Proc. Machine Learning on HPC Environments*, pp. 10:1–10:8. New York, NY: ACM Press.
17. Langou J, Langou J, Luszczek P, Kurzak J, Buttari A, Dongarra J. 2006 Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems). In *Proc. 2006 ACM/IEEE Conf. on Supercomputing, Tampa, FL*.
18. Carson E, Higham NJ. 2017 A new analysis of iterative refinement and its application to accurate solution of ill-conditioned sparse linear systems. *SIAM J. Sci. Comput.* **39**, A2834–A2856. (doi:10.1137/17M1122918)
19. Carson E, Higham NJ. 2018 Accelerating the solution of linear systems by iterative refinement in three precisions. *SIAM J. Sci. Comput.* **40**, A817–A847. (doi:10.1137/17M1140819)
20. Saad Y, Schultz MH. 1986 GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput.* **7**, 856–869. (doi:10.1137/0907058)
21. Higham NJ. 2019 Error analysis for standard and GMRES-based iterative refinement in two and three-precisions. Manchester Institute for Mathematical Sciences, The University of Manchester UK. (http://eprints.maths.manchester.ac.uk/2735)
22. Haidar A, Abdelfattah A, Zounon M, Wu P, Pranesh S, Tomov S, Dongarra J. 2018a The design of fast and energy-efficient linear solvers: on the potential of half-precision arithmetic and iterative refinement techniques. In *Computational Science—ICCS 2018* (eds Y Shi, H Fu, Y Tian, VV Krzhizhanovskaya, MH Lees, J Dongarra, PMA Sloot), pp. 586–600. Cham, Switzerland: Springer International Publishing.
23. Haidar A, Tomov S, Dongarra J, Higham NJ. 2018b Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed up Mixed-Precision Iterative Refinement Solvers. In *Proc. Int. Conf. for High Performance Computing, Networking, Storage, and Analysis (SC'18)* (Dallas, TX), pp. 47:1–47:11. Piscataway, NJ, USA: IEEE Press.
24. Buck I. 2019 World's fastest supercomputer triples its performance record. See https://blogs.nvidia.com/blog/2019/06/17/hpc-ai-performance-record-summit/ (accessed 24 June 2019).
25. Higham NJ, Pranesh S, Zounon M. 2019 Squeezing a matrix into half precision, with an application to solving linear systems. *SIAM J. Sci. Comput.* **41**, A2536–A2551. (doi:10.1137/18M1229511)
26. Higham NJ, Pranesh S. 2019 Exploiting lower precision arithmetic in solving symmetric positive definite linear systems and least-squares problems. MIMS EPrint 2019.20 Manchester Institute for Mathematical Sciences, The University of Manchester UK. (http://eprints.maths.manchester.ac.uk/2736)
27. Carson E, Higham NJ, Pranesh S. In preparation Three-Precision GMRES-based Iterative Refinement for Least Squares Problems. Manchester Institute for Mathematical Sciences, The University of Manchester UK.
28. Tomov S, Nath R, Ltaief H, Dongarra J. 2010 Dense linear algebra solvers for multicore with GPU accelerators. In *2010 IEEE Int. Symp. on Parallel Distributed Proc., Workshops and Phd Forum (IPDPSW), Atlanta, GA*, pp. 1–8.
29. Dongarra JJ, Luszczek P, Tsai YM. HPL-AI Mixed-Precision Benchmark. See https://icl.bitbucket.io/hpl-ai/.
30. Wilkinson JH. 1961 Error analysis of direct methods of matrix inversion. *J. Assoc. Comput. Mach.* **8**, 281–330. (doi:10.1145/321075.321076)

31. Higham NJ, Mary T. 2019 A new approach to probabilistic rounding error analysis. *SIAM J. Sci. Comput.* **41**, A2815–A2835. (doi:10.1137/18M1226312)
32. Blanchard P, Higham NJ, Lopez F, Mary T, Pranesh S. 2019 Mixed precision block fused multiply-add: error analysis and application to GPU Tensor Cores. Manchester Institute for Mathematical Sciences, The University of Manchester UK. (http://eprints.maths.manchester.ac.uk/2733)
33. Hatfield S, Düben P, Chantry M, Kondo K, Miyoshi T, Palmer T. 2018 Choosing the optimal numerical precision for data assimilation in the presence of model error. *J. Adv. Model. Earth Syst.* **10**, 2177–2191. (doi:10.1029/2018MS001341)
34. Palmer TN. 2020 The physics of numerical analysis: a climate modelling case study. *Phil. Trans. R. Soc. A* **378**, 20190058. (doi:10.1098/rsta.2019.0058)
35. Tintó Prims O, Acosta MC, Moore AM, Castrillo M, Serradell K, Cortés A, Doblas-Reyes FJ. 2019 How to use mixed precision in ocean models: exploring a potential reduction of numerical precision in NEMO 4.0 and ROMS 3.6. *Geoscientific Model Dev.* **12**, 3135–3148. (doi:10.5194/gmd-12-3135-2019)
36. Yang K, Chen YF, Roumpos G, Colby C, Anderson J. 2019 High Performance Monte Carlo Simulation of Ising Model on TPU Clusters. *arXiv e-prints*, p. 15. See http://arxiv.org/abs/1903.11714.
37. Hopkins M, Mikaitis M, Lester DR, Furber S. 2020 Stochastic rounding and reduced-precision fixed-point arithmetic for solving neural ordinary differential equations. *Phil. Trans. R. Soc. A* **378**, 20190052. (doi:10.1098/rsta.2019.0052)
38. Tagliavini G, Mach S, Rossi D, Marongiu A, Benin L. 2018 A Transprecision Floating-Point Platform for Ultra-Low Power Computing. In *2018 Design, Automation and Test in Europe Conf. and Exhibition (DATE), Dresden, Germany*, pp. 1051–1056.
39. Higham NJ, Pranesh S. 2019 Simulating low precision floating-point arithmetic. *SIAM J. Sci. Comput.* **41**, C585–C602. (doi:10.1137/19M1251308)
40. Wulf WA, McKee SA. 1995 Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News* **23**, 20–24. (doi:10.1145/216585.216588)
41. Graham SL, Snir M, Patterson CA (eds). 2005 *Getting up to speed: the future of supercomputing*. Washington, DC: National Academies Press.
42. Zimmer C, Atchley S, Pankajakshan R, Smith BE, Karlin I, Leininger ML, Bertsch A, Ryujin BS, Burmark J, Walker-Loud A, Clark MA, Pearce O. 2019 An Evaluation of the CORAL Interconnects. In *Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, SC'19 pp. 39:1–39:18. New York, NY: ACM.
43. Barron DW, Swinnerton-Dyer HPF. 1960 Solution of simultaneous linear equations using a Magnetic-Tape store. *Comput. J.* **3**, 28–33. (doi:10.1093/comjnl/3.1.28)
44. Anderson E, Bai Z, Bischof C, Blackford S, Demmel J, Dongarra J, Croz JD, Greenbaum A, Hammarling S, McKenney A, Sorensen D. 1999 *LAPACK users' guide*. Philadelphia, PA: Society for Industrial and Applied Mathematics.
45. Blackford LS, Choi J, Cleary A, D'Azevedo E, Demmel J, Dhillon I, Dongarra J, Hammarling S, Henry G, Petitet A, Stanley K, Walker D, Whaley RC. 1997 *ScaLAPACK users' guide*. Philadelphia, PA: Society for Industrial and Applied Mathematics.
46. Dongarra J, Gates M, Haidar A, Kurzak J, Luszczek P, Wu P, Yamazaki I, Yarkhan A, Abalenkovs M, Bagherpour N, Hammarling S, Šístek J, Stevens D, Zounon M, Relton SD. 2019 PLASMA: parallel linear algebra software for multicore using OpenMP. *ACM Trans. Math. Software* **45**, 161–1635. (doi:10.1145/3264491)
47. Grigori L, Demmel J, Xiang H. 2011 CALU: a communication optimal LU factorization algorithm. *SIAM J. Matrix Anal. Appl.* **32**, 1317–1350. (doi:10.1137/100788926)
48. Hong JW, Kung HT. 1981 I/O complexity: the red-blue pebble game. In *STOC'81: Proc. 13th Annual ACM Symp. on Theory of Computing*, pp. 326–333. New York, NY: ACM.
49. Irony D, Toledo S, Tiskin A. 2004 Communication lower bounds for distributed-memory matrix multiplication. *J. Parallel Distrib. Comput.* **64**, 1017–1026. (doi:10.1016/j.jpdc.2004.03.021)
50. Demmel JW, Grigori L, Hoemmen M, Langou J. 2008 Communication-avoiding parallel and sequential QR and LU factorizations: theory and practice. Technical Report UCB/EECS-2008-89 University of California Berkeley, EECS Department. LAWN #204.

51. Ballard G, Demmel J, Holtz O, Schwartz O. 2011 Minimizing communication in linear algebra. *SIAM J. Matrix Anal. Appl.* **32**, 866–901. (doi:10.1137/090769156)
52. Cannon LE. 1969 A cellular computer to implement the Kalman filter algorithm. PhD thesis, Montana State University.
53. Demmel J, Grigori L, Gu M, Xiang H. 2015 Communication-avoiding rank-revealing QR decomposition. *SIAM J. Matrix Anal. Appl.* **36**, 55–89. (doi:10.1137/13092157X)
54. Golub GH, Plemmons RJ, Sameh A. 1988 Parallel block schemes for large-scale least-squares computations. In *High-speed computing: scientific applications and algorithm design* (ed. RB Wilhelmson), pp. 171–179. Urbana and Chicago, IL, USA: University of Illinois Press.
55. Pothen A, Raghavan P. 1989 Distributed orthogonal factorization: Givens and Householder algorithms. *SIAM J. Sci. Stat. Comput.* **10**, 1113–1134. (doi:10.1137/0910067)
56. Demmel JW, Grigori L, Hoemmen M, Langou J. 2012 Communication-optimal parallel and sequential QR and LU factorizations. *SIAM J. Sci. Comput.* **34**, 206–239. Short version of technical report UCB/EECS-2008-89 from 2008. (doi:10.1137/080731992)
57. Ballard G, Demmel J, Grigori L, Jacquelin M, Nguyen HD, Solomonik E. 2014 Reconstructing Householder Vectors from Tall-Skinny QR. In *Proc. IEEE Int. Parallel and Distributed Processing Symposium IPDPS, Phoenix, AZ*.
58. Grigori L, David PY, Demmel J, Peyronnet S. 2010 Brief announcement: Lower bounds on communication for direct methods in sparse linear algebra. In *Proc. ACM SPAA, Santorini, Greece*.
59. Azad A, Ballard G, Buluc A, Demmel J, Grigori L, Schwartz O, Toledo S, Williams S. 2016 Exploiting multiple levels of parallelism in sparse Matrix-Matrix multiplication. *SIAM J. Sci. Comput.* **38**, 624–651. (doi:10.1137/15M104253X)
60. Hoemmen M. 2010 Communication-avoiding Krylov Subspace Methods. PhD thesis, Berkeley, CA, USA. AAI3413388.
61. Carson E. 2015 Communication-Avoiding Krylov Subspace Methods in Theory and Practice. PhD thesis, University of California at Berkeley, CA.
62. Grigori L, Moufawad S, Nataf F. 2016 Enlarged Krylov subspace conjugate gradient methods for reducing communication. *SIAM J. Matrix Anal. Appl.* **37**, 744–773. (doi:10.1137/140989492)
63. Grigori L, Tissot O. 2019 Scalable linear solvers based on enlarged Krylov subspaces with dynamic reduction of search directions. *SIAM J. Sci. Comput.* **41**, C522–C547. (doi:10.1137/18M1196285)
64. Christ M, Demmel J, Knight N, Scanlon T, Yelick KA. 2013 Communication lower bounds and optimal algorithms for programs that reference arrays—part 1. Technical Report UCB/EECS-2013-61 EECS Department, University of California, Berkeley.
65. Devarakonda A, Fountoulakis K, Demmel J, Mahoney MW. 2019 Avoiding communication in primal and dual block coordinate descent methods. *SIAM J. Sci. Comput.* **41**, C1–C27. (doi:10.1137/17M1134433)
66. Das S, Demmel J, Fountoulakis K, Grigori L, Mahoney MW. 2019 Parallel and communication avoiding least angle regression. *CoRR*. See http://arxiv.org/abs/1905.11340.
67. Solomonik E, Demmel J, Hoefler T. 2017 Communication lower bounds of bilinear algorithms for symmetric tensor contractions. *ArXiv e-prints*.
68. Ballard G, Knight N, Rouse K. 2018 Communication lower bounds for Matricized Tensor Times Khatri-Rao Product. In *Proce. IEEE Int. Parallel and Distributed Processing Symposium (IPDPS), Vancouver, Canada*, pp. 557–567.
69. Greengard L, Rokhlin V. 1987 A fast algorithm for particle simulations. *J. Comput. Phys.* **73**, 325–348. (doi:10.1016/0021-9991(87)90140-9)
70. Martinsson PG, Rokhlin V. 2007 An accelerated kernel-independent fast multipole method in one dimension. *SIAM J. Sci. Comput.* **29**, 1160–1178. (doi:10.1137/060662253)
71. Bebendorf M. 2008 *Hierarchical matrices*. Leipzig, Germany: Springer.
72. Börm S, Grasedyck L, Hackbusch W. 2003 Hierarchical matrices. (https://www.researchgate.net/publication/277293203_Hierarchical_Matrices)
73. Hackbusch W. 2015 *Hierarchical matrices: algorithms and analysis*, 3rd edn. Springer Series in Computational Mathematics. Baltimore, MD: Springer.
74. Keyes DE, Ltaief H, Turkiyyah G. 2020 Hierarchical algorithms on hierarchical architectures. *Phil. Trans. R. Soc. A* **378**, 20190055. (doi:10.1098/rsta.2019.0055)

75. Eckart C, Young G. 1936 The approximation of one matrix by another of lower rank. *Psychometrika* **1**, 211–218. (doi:10.1007/BF02288367)
76. Demmel J, Grigori L, Rusciano A. 2019 An improved analysis and unified perspective on deterministic and randomized low rank matrix approximations. Technical report Inria. See http://arxiv.org/abs/1910.00223.
77. Parlett BN. 1998 *The symmetric eigenvalue problem*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics. Unabridged, amended version of book first published by Prentice-Hall in 1980.
78. Businger PA, Golub GH. 1965 Linear least squares solutions by Householder transformations. *Numer. Math.* **7**, 269–276. (doi:10.1007/BF01436084)
79. Gu M, Eisenstat SC. 1996 Efficient algorithms for computing a strong rank-revealing QR factorization. *SIAM J. Sci. Comput.* **17**, 848–869. (doi:10.1137/0917055)
80. Grigori L, Cayrols S, Demmel JW. 2018 Low rank approximation of a sparse matrix based on LU factorization with column and row tournament pivoting. *SIAM J. Sci. Comput.* **40**, 181–209. (doi:10.1137/16M1074527)
81. Halko N, Martinsson PG, Tropp JA. 2011 Finding structure with randomness: probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Rev.* **53**, 217–288. (doi:10.1137/090771806)
82. Woodruff DP. 2014 Sketching as a tool for numerical linear algebra. *Found. Trends Theor. Comput. Sci.* **10**, 1–157. (doi:10.1561/0400000060)
83. Sarlos T. 2006 Improved Approximation Algorithms for Large Matrices via Random Projections. In *2006 47th Annual IEEE Symp. Foundations of Computer Science (FOCS'06), Berkeley, CA*, pp. 143–152.
84. Grasedyck L, Kressner D, Tobler C. 2013 A literature survey of low-rank tensor approximation techniques. *GAMM-Mitteilungen* **36**, 53–78. (doi:10.1002/gamm.201310004)
85. Hackbusch W. 2012 *Tensor spaces and numerical tensor calculus*, vol. 42. Berlin, Germany: Springer Science & Business Media.
86. Kolda TG, Bader BW. 2009 Tensor decompositions and applications. *SIAM Rev.* **51**, 455–500. (doi:10.1137/07070111X)
87. de Silva V, Lim LH. 2008 Tensor rank and the ill-posedness of the best low-rank approximation problem. *SIAM J. Matrix Anal. Appl.* **30**, 1084–1127. (doi:10.1137/06066518X)
88. Higham NJ, Mary T. 2019 Solving block low-rank linear systems by LU factorization is numerically stable. Manchester Institute for Mathematical Sciences, The University of Manchester UK. (http://eprints.maths.manchester.ac.uk/2730/)
89. Xi Y, Xia J. 2016 On the stability of some hierarchical rank structured matrix algorithms. *SIAM J. Matrix Anal. Appl.* **37**, 1279–1303. (doi:10.1137/15M1026195)
90. Dongarra J et al. 2011 International exascale software project roadmap. *Int. J. High Performance Comput. Appl.* **25**, 3–60. (doi:10.1177/1094342010391989)
91. U.S. Department of Energy. 2010 The Opportunities and Challenges of Exascale Computing. Technical report Office of Science Washington, D.C., USA.
92. Kothe D, Lee S, Qualters I. 2019 Exascale computing in the United States. *Comput. Sci. Eng.* **21**, 17–29. (doi:10.1109/MCSE.2018.2875366)
93. Kalbe G. 2019 The European approach to the exascale challenge. *Comput. Sci. Eng.* **21**, 42–47. (doi:10.1109/MCSE.2018.2884139)
94. Qian D, Luan Z. 2019 High performance computing development in China: a brief review and perspectives. *Comput. Sci. Eng.* **21**, 6–16. (doi:10.1109/MCSE.2018.2875367)
95. Sorensen B. 2019 Japan's Flagship 2020 'Post-K' system. *Comput. Sci. Eng.* **21**, 48–49. (doi:10.1109/MCSE.2018.2886646)
96. Chazan D, Miranker W. 1969 Chaotic relaxation. *Linear Algebra Appl.* **2**, 199–222. (doi:10.1016/0024-3795(69)90028-7)
97. Bethune I, Bull JM, Dingle NJ, Higham NJ. 2014 Performance analysis of asynchronous Jacobi's method implemented in MPI, SHMEM and OpenMP. *Int. J. High Performance Comput. Appl.* **28**, 97–111. (doi:10.1177/1094342013493123)
98. Chow E, Anzt H, Dongarra J. 2015 Asynchronous Iterative Algorithm for Computing Incomplete Factorizations on GPUs. In *High Performance Computing* (eds JM Kunkel, T Ludwig), pp. 1–16. Cham, Switzerland: Springer.

99. Dongarra J, Gates M, Kurzak J, Luszczek P, Tsai YM. 2018 Autotuning numerical dense linear algebra for batched computation with GPU hardware accelerators. *Proc. IEEE* **106**, 2040–2055. (doi:10.1109/JPROC.2018.2868961)
100. Whaley RC, Petitet A, Dongarra JJ. 2001 Automated empirical optimization of software and the ATLAS project. *Parallel Comput.* **27**, 3–35. (doi:10.1016/S0167-8191(00)00087-9)
101. Bouteiller A, Herault T, Bosilca G, Du P, Dongarra J. 2015 Algorithm-based fault tolerance for dense matrix factorizations, multiple failures and accuracy. *ACM Trans. Parallel Comput.* **1**, 10:1–10:28. (doi:10.1145/2686892)
102. Baboulin M, Dongarra J, Herrmann J, Tomov S. 2013 Accelerating linear system solutions using randomization techniques. *ACM Trans. Math. Software* **39**, 8:1–8:13. (doi:10.1145/2427023.2427025)
103. Mary T, Yamazaki I, Kurzak J, Luszczek P, Tomov S, Dongarra J. 2015 Performance of random sampling for computing low-rank approximations of a dense matrix on GPUs. In *Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis (SC'15)*, pp. 60:1–60:11. New York, NY, USA: ACM.
104. Inside HPC Staff. 2019 Gordon Bell Prize highlights the impact of AI. See https://insidehpc.com/2019/02/gordon-bell-prize-highlights-the-impact-of-ai/ (accessed 25 October 2019).
105. Schulthess TC, Bauer P, Wedi N, Fuhrer O, Hoefler T, Schär C. 2019 Reflecting on the goal and baseline for exascale computing: a roadmap based on weather and climate simulations. *Comput. Sci. Eng.* **21**, 30–41. (doi:10.1109/MCSE.2018.2888788)