Contents lists available at ScienceDirect

Comptes Rendus Mecanique

www.sciencedirect.com

High Performance Computing / Le Calcul Intensif

Exaflop/s: The why and the how

Exaflop/s : Pourquoi et comment

David E. Keyes [a,b,*]

[a] *King Abdullah University of Science and Technology, Thuwal 23955-6900, Saudi Arabia*
[b] *Columbia University, New York, NY 10027-4701, USA*

A R T I C L E   I N F O

Article history:
Available online 1 February 2011

Keywords:
Computer science
Exaflop

Mots-clés :
Informatique, algorithmique
Exaflop

A B S T R A C T

The best paths to the exascale summit are debatable, but all are narrow and treacherous, constrained by fundamental laws of physics, capital cost, operating cost, power requirements, programmability, and reliability. Many scientific and engineering applications force the modeling community to attempt to scale this summit. Drawing on vendor projections and experiences with scientific codes on contemporary platforms, we outline the challenges and propose roles and essential adaptations for mathematical modelers in one of the great global scientific quests the next decade.

© 2010 Published by Elsevier Masson SAS on behalf of Académie des sciences.

R É S U M É

On peut discuter du meilleur chemin vers les sommets de l'exascale mais tous sont traitres et étroits, conditionnés par les lois fondamentales de la physique, les couts d'investissement et de fonctionnement, la consommation électrique, la facilité de programmation et la fiabilité. Pour la plus part des applications la science et l'ingénierie, la communauté des modélisateurs sera forcée de s'adapter pour gravir ce sommet. Partant des prévisions des constructeurs et de notre expérience des codes scientifiques sur les plateformes actuelles, nous tracerons les grandes lignes du défit exascale et proposerons des scénarios et des adaptations aux numériciens pour ce grand challenge scientifique de la décennie.

© 2010 Published by Elsevier Masson SAS on behalf of Académie des sciences.

1. Introduction

Sustained floating-point computation rates on real applications, as tracked for instance by the ACM Gordon Bell Prize, increased by three orders of magnitude from 1988, when 1 Gigaflop/s was first reported on a structural simulation, to 1998, when 1 Teraflop/s was first achieved, and by another three orders of magnitude to 2008, when 1 Petaflop/s was attained on a molecular dynamics simulation. Computer engineering provided only about two orders of magnitude of improvement for individual cores over that two-decade period; the remaining four orders of magnitude in performance came from concurrency, which is approaching one-million-fold on today's highest-end platforms. Algorithmic improvements contributed meanwhile to making the average flop more valuable scientifically. As the semiconductor industry now slips relative to its own roadmap for silicon-based logic and memory, concurrency, especially on-chip many-core concurrency of either homo-

[*] Address for correspondence: King Abdullah University of Science and Technology, Thuwal 23955-6900, Saudi Arabia.
E-mail addresses: david.keyes@kaust.edu.sa, david.keyes@columbia.edu.

1631-0721/$ – see front matter © 2010 Published by Elsevier Masson SAS on behalf of Académie des sciences.
doi:10.1016/j.crme.2010.11.002

geneous or heterogeneous type, will play an increasing role in the next few orders of magnitude, to arrive at the ambitious target of 1 Exaflop/s, extrapolated for 2018.

An important question is whether today's best algorithms are efficiently hosted on the exascale hardware the scientific community is likely to receive, and how much co-design of algorithms and architecture will be required. Simulation-based scientific discovery and engineering design have long hotly pursued the extreme scale of computing, not for its own sake, but for progress on applications with requirements for spatial resolution and temporal cycles. From the perspective of applications, numerous reasons for an insatiable appetite are summarized in [1]. We mention here: resolution, fidelity, dimensionality, artificial boundaries, multiphysics, parameter inversion, optimal control, and uncertainty quantification. There is intrinsic interest in passage to the exascale for individual codes in these driving scientific and technological campaigns. There is additional interest in their simultaneous combination. There is still further interest in solving these coupled complex systems repeatedly, as is required typically to gain scientific understanding from simulation, in the form of solving inverse problems, or performing sensitivity analysis or uncertainty quantification.

To illustrate these pushes for increased computational resolution and processing, consider, for instance, how each of these might be manifest in a petroleum production company:

1. *Better resolve a model's full, natural range of length or time scales.* Better accuracy in the evolution of models of an oil-producing reservoir requires that the reservoir be discretized into more horizontal cells and more vertical layers. Geophysicists would today like to achieve 25 m or less horizontal spacing between mesh cells, with denser mesh clustering around injection and production wells. Tomorrow they will want more. Vertical resolution requirements are determined largely by local geology. A billion mesh cells do not yet accommodate resolution requirements for the world's largest reservoirs.
2. *Accommodate physical effects with greater fidelity.* The "black oil" assumption fails to take into account the relative composition of different molecular compositions of a true reservoir, as well as the different phases present. The number of degrees of freedom present at each cell must be expanded to distinguish between these fractions.
3. *Allow the model degrees of freedom in all relevant dimensions.* All practical reservoirs are finite and irregular in their geometry. Simulations based on two-dimensional or one-dimensional models can produce scientific insight but not predictive results. In this case, all three space dimensions plus time must be resolved.
4. *Better isolate artificial boundary conditions.* Rarely is it possible to isolate a system for analysis through the accurate specification of boundary conditions at its domain surface. However, one can apply artificial boundary conditions sufficiently far away from the subdomain where one wishes to observe the solution that the artificial treatment does not affect the solution over some desired time interval, which is determined by the most stringent signal transit time.
5. *Combine multiple complex models.* Producing a reservoir induces changes to the surrounding geology, such as subsidence or minor quakes, which are interesting candidates for modeling in their own right, and coupling is two-way. Unifying the simulations of the multiple physics into one model is ultimately critical to yielding accuracy improvements.
6. *Solve an inverse problem or perform data assimilation.* Reservoir models are parameterized with coefficients, forcing terms, and initial and boundary conditions that can never perfectly be known. However, petroleum production companies maintain wellhead records that can be compared with simulation results to better estimate unknown parameters in the model and "nudge" the simulation towards reality.
7. *Perform optimization or control.* For an engineer, the ability to model accurately the result of a scenario is just a prelude to optimization or control of the system. Modelers would like to put the predictive model inside a control loop and select a strategy for injection and pumping in the hundreds or thousands of wells in a given reservoir.
8. *Quantify uncertainty.* Given the many unresolvable uncertainties in program inputs, a modeler would like to bound the error in the outputs in terms of errors in the inputs. This again requires putting the forward model inside a loop. A brute-force means of doing this is through statistical ensembles, though there are potentially superior means through the introduction of additional variables.

The memory- and cycle-consuming agendae listed above only address the obvious opportunities to exploit the next boost of a factor of a thousand in computing capacity in modeling a single reservoir. Oil companies are vast, with dozens of reservoirs "upstream" and many refineries and transportation systems "downstream." Companies may produce many different refined products whose values vary over time, and function under many time-varying constraints for meeting product supply. They must seek to maximize profit while satisfying output demands in many different product streams. This is a system with tens of thousands of "valves" (literal and figurative, like workforce and other controls) that need to be continuously scheduled. Mathematically, this is a massive, nonlinear and possibly nonrobust constrained optimization problem – insatiably power hungry. Such problems are too complex for a self-contained, self-consistent theory. On the other hand, they are unsuitable for experiment, because the experiment (e.g., exploiting a reservoir) can be done but once. Engineering heuristics are useful (and used!), but gone are the days when employees spent a career understanding a single reservoir. Furthermore, the external forcings (e.g., the world economy) change frequently, making history less useful. Simulation is a unique tool for exploring scenarios experimentally in a virtual world. Data mining and machine learning may be even more useful tools than simulation in the future.

The perspective of this article is that applications are given (as function of time). Architectures are also given (as function of time), influenced, to be sure, by what is possible and required in terms of scientific and engineering applications but mainly commercially driven. Algorithms and software must be adapted or created to bridge to hostile architectures for the sake of the complex applications. This is as important as ever today, with transformation of Moore's Law from speed-based to concurrency-based, due to power considerations. Scalability is therefore important, but other stresses arise when on-chip memories are shared. Knowledge of algorithmic capabilities can usefully influence the way applications are formulated and the way architectures are constructed. Knowledge of application and architectural opportunities can usefully influence algorithmic development.

Several formulations of scientific and engineering models require scaling: partial differential equations (PDEs) for continuous problems, whose fields represent primary observables of the physical universe; integro-differential equations for probability densities, from which the observables can be obtained by moments; systems of interacting particles; agent-based models; and so forth. In this article we focus on PDEs. While not spanning the full space of scientific computing, PDEs make up the core of the applications that drive computational scaling today, and their parallel implementations share many relevant features with the other models. They involve both local (explicit) and global (implicit) coupling. Their data structures are increasingly unstructured in practice. Their work per degree of freedom is increasingly nonuniform. Applications that involve PDEs are increasingly multi-phase computations, meaning that there are multiple global data structures that need to be partitioned and load-balanced, and the couplings between the multiple distributed structures are not possible to load balance in a way that all data dependencies between them are local. Therefore, PDEs are challenging representative objects of study in the exascale quest.

2. State of the art for PDE-based simulations

Partial differential equation-based applications today scale to the edge of available platforms, using primarily bulk synchronous [2], MPI-based implementations on statically or quasi-statically load-balanced partitions. Such codes are regularly entered into the Gordon Bell Prize competition, and they occasionally win the "special" prize, on the basis of their scaling, though they generally do not achieve a high percentage of peak flop/s [3]. Flop/s rates are limited by the low memory bandwidth of current hierarchical memory systems, in which access to DRAM is at least two orders of magnitude longer than a processor cycle, yet elements of (typically) the largest workingset in the problem, a Jacobian matrix, have poor locality, and need to be regularly fetched afresh from DRAM.

For the earliest exascale driver applications, we naturally look to today's petascale applications. (We anticipate that entirely new types applications will join them as creative scientists and engineers are afforded architectural headroom, but petascale applications are already ripe – a factor of one thousand in capability is actually a modest factor of ten improvement in resolution when cube-rooted into three space dimensions.) Today's petascale applications are built on a large number of software toolkits, including: geometric modelers, meshers, discretizers, partitioners, solvers and integrators, systems for mesh and discretization adaptivity, random number generators, libraries of subgridscale physics models and constitutive relations, uncertainty quantification, dynamic load balancing, graphical and combinatorial algorithms, and data compression methods.

Code development effort at the petascale relies on another fleet of software toolkits, such as configuration systems, compilers and source-to-source translators, messaging systems, debuggers, profilers, simulators and regression test harnesses. Finally, production use of these petascale applications relies on dynamic resource management, dynamic performance optimization, authentication systems for remote access, I/O systems, visualization systems, data miners, workflow controllers, and frameworks. These toolkits will all need to be supported on emerging exascale machines, where they will need to be supplemented beyond their counterparts today by additional tools, such as: fault monitoring, fault reporting to the application, and fault recovery.

Three general characteristics of PDE-based applications that are relevant to challenges that await at the exascale are summarized here:

Fully implicit coupling. Temporally implicit, fully coupled formulations are often preferred for multirate applications whose dynamics of interest are in scales that are much slower than the fastest wavespeeds in the problems. High-order temporal discretizations are useless in the presence of loosely coupled low-order temporal operator splitting, which is one factor that drives scientists towards full coupling. Also, stability theory for loose coupling of nonlinear problems is incomplete, at best. Synchronization at every (macro) time step at which we wish to observe the solution cannot be avoided, and more frequent synchronization is built into today's algorithms (artifactually and removably in some cases). Implicit methods potentially improve on explicit by offering lots of useful work between synchronizations, in effect reducing their frequency. They advance to a given physical time on a much smaller "budget" of synchronizations.

Lack of amenability to time parallelism. Scientists often express the desire for strong speed-up in order to reduce the execution time of large-scale simulations, and sometimes propose parallelization in the temporal dimension of the model in pursuit of this aim. Temporal parallelization is an important research topic in its own right, at many scales smaller than exascale. However, opportunities for parallelization in time are limited at the exascale by memory considerations. Parallelization in time typically requires simultaneous storage of multiple copies of the spatial discretization at successive temporal stages. We see below that storage of even one copy of the spatially complete state of the system is challenging at the architectural proportions of the exascale. An architectural pressure of the exascale is to give ground elsewhere to save memory; we do not have excess memory to exploit on behalf of other objectives.

Adaptivity. Solution-based adaptivity, which is highly desirable for numerical efficiency, is another challenge at the exascale, at least the way adaptation is practiced today. The adaptive step typically performs it own synchronization and must be followed by a dynamic *in situ* load balance, since the classical SPMD bulk synchronous paradigm requires statically balanced work per physical degree-of-freedom per major iteration.

3. Architectural constraints

The space of architectural possibilities is limited by laws of physics (speed of light, heat capacity of materials, manufacturing limits in lithography, etc.), capital cost, operating cost, power requirements, programmability, and reliability. It would be the proper scope of a much different article to probe how each limits different aspects of hardware design. Individual processor cores, memory units, and communication hardware are subject to their own physical and cost constraints that lead to optimized components. We could raise here a higher level question: In what proportions and through what logical topologies and physical geometries should these components be combined? Algorithm designers with a practical motivation need not explore this space in full because it is relatively clear from industry projections that the first exascale systems will be close to a certain point that we may call the kilo-core mega-node giga-Hertz prototype. The prefixes multiply to 10^{18}, not taking into account multiple functional units firing on each clock, which is at best very small factor. We recap some basic architectural constraints; see, e.g., [4] for a recent survey of the landscape and [5] for implications at the exascale.

Capital cost of acquisition is a first-class constraint, in practice, and its greatest effect is to limit the total storage, to, say, $100M of the cost of a $200M exascale system. How much storage can be purchased for this budget in 2018 will ultimately be determined by the cost of 8 GB, 16 GB, or tighter packing modules, but it is unlikely to be more than 128 PB, leaving exascale machines memory-thin, relative to the Amdahl-Case Rule [6], by about an order of magnitude.

Power cost of operation, the availability of sufficient power at any cost, and the physical ability to dissipate consumed power make up another primary constraint. Exascale systems cannot realistically be expected to consume more than about 20 MW on the computation alone (irrespective of cooling, etc.). This is approximately the power required by a town of 14 K inhabitants in any OECD country, where the average continuous electrical power consumption is 1.4 KW per person. The energy cost per bit transferred translates this power envelope to a number of bytes per second that is likely to be less than one per flop/s. Even this miserable bandwidth would require enhanced memory with transfer costs of approximately 4 pJ/bit, whose most optimistic cost by 2018 is likely to bust the capital budget estimates of the preceding paragraph. Driving down the capital budget for the memory while holding to the 20 MW power budget, by exploiting less power efficient designs, such as a 30 pJ/bit target (which will be about an order of magnitude cheaper), costs another order of magnitude of memory bandwidth performance, down to about 0.1 Byte/flop. This is a drastically low level for known algorithms and getting even to this minimal provision requires memory technology breakthroughs relative to the 100 to 150 pJ/bit cost today.

Kilo-core mega-node giga-Hertz machines equipped with about 100 PB of memory will likely be the first exascale systems built, because they will rely on components that are manufacturable in sufficient volume to pay for their engineering, and because the trade of high concurrency for a low frequency within a constant product is one that minimizes power. Power consumption rises approximately as the cube of the frequency, so that four cores running at 1 GHz are sixteen-fold cheaper to power than one core running at 4 GHz.

If such systems are cheaper to power, it is unfortunate that they may be hard to exploit at full processor potential. This is primarily due to the fact that memory bandwidth already limits today's low core-count nodes to less than 10% of peak on most applications like PDEs, whose dominant kernels offer little cache reuse. (This is true, for instance, of applying stencil operations explicitly or performing a sparse matrix–vector multiply, which is practically the same task.) The sparse matvec is not the only kernel in a PDE computation, and indeed there are others, such as evaluating the fluxes that go into setting the matrix elements, that are capable of exploiting many more flops per memory reference. Processors are cheap and (relative to memory) small in chip area and low in power, so there is little harm in having them in excess most of the time, but the opportunities for exploiting this main new source for performance, namely the profusion of cores per node, are undemonstrated for most applications. Counting flops per second as a fraction of peak is a common figure of merit, but this must be dismissed if vendors over-provision flop/s relative to memory bandwidth as a design choice.

4. Some implications for algorithms

Just as the exascale architectural peak can be approached as the product of three factors representing: the on-node concurrency, the number of nodes, and the frequency of each core, exploiting the billion-fold thread concurrency of the exascale can be approached through three factors: strong scaling within the fixed amount of memory per node, weak scaling in the number of nodes, and workload or ensemble scaling beyond the individually scaling independent executions. While there is potential opportunity for combining today's individually high capability simulations into complex simulations, there is no silver bullet for merging the data structures of the separate applications. The data copying inherent in the code coupling will likely inhibit exploitation of the apparent concurrency opportunities.

Prior to possessing exascale hardware, users can prepare themselves for the strong scaling required by exploring new programming models on many-core and heterogeneous nodes. Attention to locality and reuse is valuable at all scales and will produce performance paybacks today and in the future. New algorithms and their data structures can be explored under the assumption that flop/s are cheap and moving data is expensive. Benefits of two-level parallelism can be seen today in kernels such as FFT and in fuller applications. However, on-node shared-memory parallelism is not easy to demonstrate yet in PDE-based applications at the "kilo-core" scale.

A compelling implication of the two primary hardware constraints – memory pressure and memory bandwidth pressure – is to walk away from the numerically convenient assumption of default double precision. An exciting prospect in programming exascale hardware is to exploit mixed precision floating point arithmetic, striving to use the lowest required in a local tasks to achieve a given global accuracy outcome. The power and speed benefits of lower precision have been dramatically demonstrated on heterogeneous processors, such as those built from the early single-precision GPGPUs. The promises include reduced execution times, reduced communication times, and reduced power consumption. Such techniques are not unknown in petascale practice. In fact, for over a decade they have been in practice in DOE warhorse codes such as PETSc [7], in which the preconditioner elements are stored and communication in single precision by default, whereas the Krylov computation is performed in double precision. High precision is needed in the orthogonalization steps in this and many other algorithms. Algorithms such as iterative refinement, which make use selective use of higher precision, are classical in numerical analysis. The key is to reformulate algorithms to operate on corrections, rather than cumulative solutions. The solutions must be stored in desired precision, but the corrections, which ultimately affect only the lower bits of the solution, can be calculated in lower precision. Fully exploiting reduced precision may require sorting operands by magnitude in commutative operations.

Resiliency, or fault-tolerance, is an important consideration that also demands much research in going from petascale to exascale; however, it should not be showstopper despite the fearsome factor of a thousand times more threads on the critical path of the computation. First, the number of physical nodes is not much greater since most of the parallelism occurs within a node. Hardware failure rates, after the chips are burned in and certified following manufacture, do not grow as strongly with the complexity of the chip as with the number of sockets. More importantly, in the face of decreasing time between failures, driving physical applications generally possess a time-dependent workingset that varies in size periodically and predictably, reaching minimal size at each timestep or evolutionary step towards equilibrium, when the physical state is updated. Application programmers can be called upon to identify this minimal size workingset and remove the burden of complete reconstruction of arbitrary state from the operating system. The loss, upon restart, of a vast number of workspaces that PDE-based codes carry around, such as tables for equation of state on the physics side or stored and factored Jacobian matrices on the numerical side, can be tolerated. They are ephemeral and can be regenerated from the minimal physical state variables, which are dumped often for visualization/analysis. To be sure, they can be reused with efficiency, and are in good implementations, but in the case of failure, the computation can be rolled back to the last dumped state and restarted with some overhead of recalculation. Some faults like soft memory errors occur in parts of the code that are not on the critical path of state evolution, but carried to improve performance or to aid interpretation. While it is inefficient to lose them, and while their loss may cost a frame of a visualization or an error bar in an uncertainty quantification, or an optimization step that must be discarded along the way, the overall forward model may remain in tact, and not all errors must trigger restart. For this advanced form of application-mediated resiliency, significant new tools for monitoring faults and their degree of severity and reporting them to the user are required, and users must be willing to enter into the recovery process in a dance of co-design that is unprecedented to date at the petascale.

As discussed in Section 3, percentage attained of peak flop/s is a counter-productive metric, since it penalizes for over-provisioning something cheap. Percentage attained of the instantaneously limiting resource, typically memory bandwidth, is the most interesting figure of merit for a given execution of the "right problem." The "right problem" is the one that provides the requisite accuracy with the requisite confidence at some combination of lowest energy and shortest execution time. It is a grand challenge of numerical analysis to equidistribute the multitudinous errors in a computation as to achieve the target result without "oversolving" or "overresolving" some contributing step. Research in the propagation of numerical errors throughout an end-to-end computation, always timely in numerical analysis at any scale, takes on a greater importance at the exascale, since the balance of resources required to fit a computation into a given memory and energy budget is increasingly delicate.

5. From the petascale to the exascale: relaxing synchrony

In contemplating the direct map of petascale applications to the exascale, the following issues present themselves as the greatest challenges. Each one of them is an exacerbation of an issue that already exists at the petascale:

- Poor scaling of global reductions due to internode latency.
- Poor performance of sparse matrix–vector multiply due to shared intranode bandwidth.
- Poor scaling of coarse grid solutions in a hierarchical algorithm due to insufficient concurrency.
- Difficulty of understanding and controlling vertical memory transfers.

Passage to the exascale introduces interesting incentives for taking application codes beyond their nearly universal roots in classical bulk synchronization protocols [2]. There are sources of nonuniformity from the hardware and the systems software that are too dynamic and too unpredictable or difficult to measure to be consistent with bulk synchronization. On the hardware side, these include: manufacturing nonuniformities, dynamic power management, and runtime component failure.

On the systems software, they include: OS jitter, software mediated resiliency, and TLB/cache performance variations. Network nonuniformities can arise from either hardware or software reasons. The nonuniformities that we have traditionally sought to balance in the numerical models, including: physics at gridcell scale (e.g., table lookup, equation of state, external forcing), discretization adaptivity, solver adaptivity, precision adaptivity, are – from an operational viewpoint – interchangeable with these other sources. We note that any efforts that we need to make at the exascale to accommodate the union of these sources of nonuniformity will help us at the petascale and the terascale, where the systems-related concerns are much smaller, but the opportunities available if we could better tolerate model-related nonuniformities are extensive!

Our "working factorization" of hardware resources that achieves exascale simulation is billion-way concurrency with gigaHertz cores. Compared to the (nearly) million-way parallelism of gigaHertz processors that achieves petascale simulation today, the main development that must be accomplished is to expand today's million-way flat parallelism at the node level with thousand-way parallelism within a node. This offers a pseudo-evolutionary path for applications, in that MPI-based legacy code will still be usable on the million nodes if bulk SPMD synchrony can continue to be observed. Changes would in this case be mainly within node, where we will need to evolve thousand-way parallelism: "MPI+X". Under the MPI+X model, the principal challenges from peta to exa are within the node, and the burden of migration from a handful to a thousand process threads per node is shared by the marketplace, at all incremental scales of node aggregation. Under this incremental hardware path to the exascale, loosely speaking, the algorithmic path from peta to exa preserves the focus on weak scaling between nodes and adds (mostly) strong scaling within nodes. There is sufficient concurrency in tasks like PDEs for thousand-fold strong scaling if sufficient memory is available; see, e.g., [8], which was mediated by MPI alone. A key consideration in expanding from today's million-fold thread concurrency to billion-fold is in the global reduction. The synchronization cost of global reductions propagates to the thousand intranodal threads with a much smaller coefficient than the MPI-mediated internode part. Dynamic load balance through work stealing is relatively more sufferable in this architecture, since remote DRAM access is not inordinately faster than local DRAM access. The cost is in the vertical and the horizontal is not significantly greater.

Domain decomposition is the weak-scaling method of choice, for PDE-derived discrete systems. It allows asymptotically linear scaling of nodes with problem size, per iteration, given log-diameter global reductions [9]. It may have an asymptotically constant number of outer nonlinear iterations, given a superlinearly convergent inexact Newton method. Under appropriate physics-based, Schwarz-based, or multigrid-based preconditioned Krylov iteration, it may have an asymptotically constant number of inner iterations in solving the linear systems that arise at each outer Newton iteration. This asymptotically optimal scaling is nontrivial to achieve at all three levels – the per-iteration constant scaling, the constant number of linear iterations, and the constant number of nonlinear iterations. However, such scaling has been approached on a number of applications that make effective use of today's petascale machines. Continued research is necessary on preconditioning that extends such results from where the beacon of theory reveals the path to more and more complex mathematical operators.

One of the most important requirements of the exascale for applications is the reduction of synchronization. While the elimination of periodic synchronization appears impossible, application writers and library designers can write code in styles requires significantly less artifactual synchronization than today. This can be illustrated with recourse to an implicit loop in a typical PDE-based application. On the critical path of such a solve is a loop that consists of forming a global residual, performing a solve to obtain a correction, bounding the step of the correction, and updating physical state. This loop, which embeds various global reductions, cannot easily or apparently be avoided. However, we often insert into this path things that could be done less synchronously, because we have limited language expressiveness to indicate that their importance is secondary. For instance, Jacobian refresh and preconditioner refresh can be taken off the path. In the best implementations, they are already done infrequently relative to use, to amortize their relative high cost of formation and application. They must be refreshed for theoretical attainment of convergence in nonlinear problems, but if some processors are in arrears heading to the next synchronization, they can be deferred. This is most easily done if the code to refresh them is isolated from the code that uses them, and called when cycles are available, so that critical path progress does not stall for convergence reasons. The main distinction between these tasks and the tasks on the path is that they are very time-consumptive and their load can be redistributed dynamically without halting the critical tasks. The same is true for many other tasks that are performed on the critical path today, but could be done less frequently. This includes convergence testing, itself. (Many iterations could be performed between tests, at the cost of an occasional unnecessary iteration.) Similar lower priority status can be accorded subthreads that perform constitutive updates, algorithmic parameter adaptation, I/O, compression, visualization, data mining, uncertainty quantification, and so forth.

In addition, in pursuit of synchronization reduction, "additive operator" versions of sequentially "multiplicative operator" algorithms are often available. Such algorithms have sometimes received a bad rap historically, e.g., "chaotic relaxation" as proposed by Chazan and Miranker in 1969 [10], for instance. In retrospect, these were ahead of their time in that the hardware of the day did not drive the demand or hold out the opportunity that it does today. It has been shown that asynchronous additive methods can sometimes be made virtually as good as their multiplicative cousins, for instance "AFACx" versus "FAC" [11]. Preconditioning subdomain blocks can be done with variable count to fill a given synchronization interval, provided that an accommodative Krylov method, such as FGMRES, is employed [12].

To take full advantage of such synchronization-reducing algorithms, modelers evidently need to develop greater expressiveness in scientific programming. They need to be able to create separate subthreads for logically separate tasks, whose priority is a function of algorithmic state, not unlike the way a time-sharing operating system works. Some software paradigms may already exist that can be put to use in this way, for instance the Asynchronous Dynamic Load Balancing (ADLB) construct of Lusk et al. [13], or prioritized forms of tuple-space programming. In adopting such programming styles, we may need to introduce "soft barriers" to prevent state from getting too stale. Exascale programming will require prioritization of critical path and noncritical tasks, adaptive DAG scheduling of critical path tasks, and adaptive rebalancing of all tasks, with the freedom of not putting the rebalancing of noncritical-path tasks on the path, itself.

Examination of exascale programming models should have several facets, among which many related to the exposing and exploiting of concurrency are already standard, and are omitted here for brevity. The key new concepts to be researched include:

1. For each problem formulation and each concurrency-motivated decomposition, isolate the tasks that can be deferred from the tasks that are on the critical path and determine or estimate the costs associated with deferring tasks. Such costs can include memory that is unreleased because data is still required, convergence rates that are degraded relative to operating with fresher data, and so forth. This is a partly analytical and partly experimental study that is performed off-line.
2. Use the decomposition and cost estimates to determine dynamically the priority for execution of tasks within a node-based process. This is a possibly complex operation that is studied off-line as a research task in its own right, but implemented in the run-time environment, as the responsibility of the application programmer.
3. Replace the procedural loop of a standard simulation code with a directed acyclic graph and a data flow execution model. The data flow model is relaxed so that tasks can be assigned for processing even if some of their lower-priority input arguments (such as the factors of a preconditioner matrix block) are not fresh. In order to exploit opportunities of a dynamically scheduled execution environment to enhance robustness, copies of each task's input arguments can be maintained until the task is complete, allowing re-issue of tasks that fail.

Algorithmic research can pay dividends well beyond research into novel programming styles and implementations. For instance, communication-reducing algorithms [14] have been demonstrated for solvers that trade the frequency of communication against memory costs and extra flops to overcome the potential loss of stability of such methods. Backing up still further, of course, better mathematical formulations may be available. A trend to be exploited where possible, for instance, is the replacement of partial differential equations with integral equations, or of preconditioners for partial differential equations with integral equation-mediated operations.

6. Necessity of co-design

Approximately 98% of microprocessors are used in embedded systems, in which the hardware and software are designed together. In contrast, scientific programmers emphasize portability – nearly hardware-independent codes on general-purpose processors. Co-design of architecture, software, and algorithms is emerging as a necessity to negotiate the physical, power, and cost limitations of exascale systems [5,15]. Though it inveighs against the computer science aesthetic of "separation of concerns," the nose of the camel of co-design has poked inside our tents for two decades. Message-passing programmers, in effect, write a piece of the runtime system when they write software libraries and applications. The attitude and aptitude required for co-design is therefore not entirely unfamiliar nor unwelcome. However, it greatly expands in scope when the hardware design, itself, becomes involved. As in the case of scientific software toolkits based on message-passing, we aim to isolate application programmers from many hardware and software architectural details of exascale execution, just as we do today from message-passing details at the petascale.

Software engineering practices that have already become well established at the petascale for cyberinfrastructure in common use today should remain the same as we transition to the exascale, to leading order. However, the opportunities of co-design may force some compromises. One of the most important of these practices is multilayer software design. Successful software is multilayered in its accessibility. The outer layer, which is accessible to all users, is an abstract interface featuring language of application domain, hiding implementation details, with conservative parameter defaults. For instance, a vector should be represented and manipulated as an element of a Hilbert space, without regard for how it is implemented in an array data structure partitioned across multiple memory systems. The goals of accessing the software at this layer are robustness, correctness, and ease of use. The middle layers, which can be opened up to experienced users through handles to the underlying objects, provide a rich collection of state-of-the-art methods and data structures, exposed upon demand, and variously configurable. The goals of access are capability, algorithmic efficiency, extensibility, composability, and comprehensibility of performance and resource use through profiling. The inner layer is intended for developer access only. It includes support for variety of hardware and software execution environments. The goals of access are portability and implementation efficiency. For the sake of removal of data copies and data structure reformatting between independently written code modules, modularity will probably be greatly inhibited as we approach the exascale. Indeed, much of the scientific software engineering philosophy espoused in [16] may need to be sacrificed.

7. The path forward

In the cross-hairs of the international exascale computing campaign are the mathematics of algorithms that must be implemented on exascale hardware and on programming models capable of expressing existing and new algorithms to the much less schedulable and more distributed hardware that is emerging. Technical goals include: identification of new concepts and abstractions that relax conservative and unrealistic assumptions of today's procedural implementations of simulations and transcend individual applications, and quantification of trade-offs of resources required (processing rate, words of memory, precision of words, interprocessor and interlevel bandwidth and latency, degrees of connectivity, etc.) for various important "dynamical cores." The outcomes should be useful to applications designers and to architects, but will primarily focus analysts on new problems for which the classical complexity questions of numerical analysis are not yet well answered.

The net result of programmer-introduced and architecture-introduced nonuniformity is that fine load balancing will either be impossible to achieve, too expensive to maintain, or too complex to code for. In the worst case, in order to make scaling feasible, one can imagine spending a comparable or greater amount of execution time engaged in dynamic rebalancing to that in useful state-advancing execution.

The evolution of today's simulation codes from the infra-petascale to the ultra-exascale need not introduce radical ideas to the world of computing. Time-sharing operating systems and job queuing systems, for instance, provide analogs for the decomposition and scheduling of subtasks of different priorities, and directed acyclic graphs are commonplace in the scheduling of algorithmic steps in many contemporary codes, where time-dependent data dependencies must be taken into account. They are generalizations to mainstream procedure-based simulations of embarrassingly parallel execution environments, such as MapReduce [17]. Since we are concerned with partitioned and distributed data structures such as graphs and lists for mathematical objects such as meshes or particle ensembles, this topic is widely applicable. In the context of scientific simulations, they may appear radical but we build upon a legacy of expertise in other domains of computer science. We conclude our launch of the quest of the exascale summit with the words of an earlier decadal quest [18]:

"We choose to do [these] things, not because they are easy, but because they are hard, because that goal will serve to organize and measure the best of our energies and skills, because that challenge is one that we are willing to accept, one we are unwilling to postpone, and one which we intend to win..."

References

[1] D.E. Keyes, Partial differential equation-based applications and solver algorithms at extreme scale, Int. J. High Perf. Comput. Appl. 23 (2009) 366–368.
[2] L. Valiant, A bridging model for parallel computation, Comm. ACM 33 (1990) 103–111.
[3] W.D. Gropp, D.K. Kaushik, D.E. Keyes, B.F. Smith, High performance parallel implicit CFD, Parallel Comput. 27 (2001) 337–362.
[4] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, K. Yelick, A view of the parallel computing landscape, Comm. ACM 52 (2009) 56–67.
[5] J. Dongarra, P. Beckman, et al., The international exascale software project roadmap, Int. J. High Perf. Comput. Appl. 25 (2011), in press.
[6] G. Bell, Great and big ideas in computer structures, in: Mind Matters: A Tribute to Allen Newell, 1996, pp. 189–218.
[7] S. Balay, K. Buschelman, W. Gropp, D. Kaushik, M. Knepley, L.C. McInnes, B.F. Smith, H. Zhang, PETSc Web page, http://www.mcs.anl.gov/petsc/, 2010.
[8] C. Burstedde, O. Ghattas, M. Gurnis, G. Stadler, E. Tan, T. Tu, L.C. Wilcox, S. Zhong, Scalable adaptive mantle convection simulation on petascale supercomputers, in: SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, IEEE Computer Society, 2008, pp. 1–15 [Gordon Bell Prize finalist.].
[9] D.E. Keyes, How scalable is domain decomposition in practice? in: C.-H. Lai, et al. (Eds.) Proceedings of the 11th Intl. Conf. on Domain Decomposition Methods, 1998, pp. 286–297.
[10] D. Chazan, W.L. Miranker, Chaotic relaxation, Linear Algebra Appl. 2 (1969) 199–222.
[11] B. Philip, B. Lee, S. McCormick, D. Quinlan, Asynchronous fast adaptive composite grid methods: Theoretical foundations, SIAM J. Numer. Anal. 42 (2004) 130–152.
[12] T.P. Collignon, M.B. Van Gijzen, Fast iterative solution of large sparse linear systems on geographically separated clusters, Int. J. High Perf. Comput. Appl. (2010), in press.
[13] E.L. Lusk, S.C. Pieper, R.M. Butler, More scalability, less pain: A simple programming model and its implementation for extreme computing, SciDAC Rev. 17 (2010) 30–37.
[14] J. Demmel, M. Hoemmen, M. Mohiyuddin, K. Yelick, Communication-optimal iterative methods, J. Phys. Conf. Ser. 180 (2009) 012040.
[15] D. Brown, P. Messina, P. Beckman, D. Keyes, J. Vetter, Crosscutting technologies for computing at extreme scale, U.S. Department of Energy, 2010, in press.
[16] D.E. Keyes, Letting physicists be physicists, and other goals of scalable solver software, in: A. Deane (Ed.), Proceedings of Parallel CFD'05, Elsevier, 2005, pp. 51–75.
[17] J. Dean, S. Ghemawat, MapReduce: Simplified data processing on large clusters, in: OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December 2004.
[18] J.F. Kennedy, We choose to go to the Moon. Speech delivered at Rice University, Houston, Texas on 12 September 1962.