PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A

rsta.royalsocietypublishing.org

Review

Cite this article: Giles MB, Reguly I. 2014 Trends in high-performance computing for engineering calculations. *Phil. Trans. R. Soc. A* **372**: 20130319.
http://dx.doi.org/10.1098/rsta.2013.0319

One contribution of 13 to a Theme Issue 'Aerodynamics, computers and the environment'.

Subject Areas:
computer modelling and simulation

Keywords:
high-performance computing, multicore, manycore, GPU, accelerator, energy efficiency

Author for correspondence:
M. B. Giles
e-mail: mike.giles@maths.ox.ac.uk

Trends in high-performance computing for engineering calculations

M. B. Giles[1] and I. Reguly[2]

[1]Mathematical Institute, and [2]Oxford e-Research Centre, University of Oxford, Oxford, UK

High-performance computing has evolved remarkably over the past 20 years, and that progress is likely to continue. However, in recent years, this progress has been achieved through greatly increased hardware complexity with the rise of multicore and manycore processors, and this is affecting the ability of application developers to achieve the full potential of these systems. This article outlines the key developments on the hardware side, both in the recent past and in the near future, with a focus on two key issues: energy efficiency and the cost of moving data. It then discusses the much slower evolution of system software, and the implications of all of this for application developers.

1. Introduction

The progress in high-performance computing (HPC) over the past 20 years is really remarkable. Figure 1 shows the performance of the Top500 list of supercomputers [1]. Although there are new more appropriate performance tests being developed [2], currently performance is measured on a dense matrix–matrix multiplication test which has questionable relevance to real applications. In particular, it is compute-dominated and does not exercise the memory subsystem as much as most real applications. Nevertheless, the figure captures the incredible advances in processor technology and all of the accompanying aspects of computer system design, such as the memory subsystem and networking, which have delivered a factor of 2 increase in performance every 18 months.

It appears that we can safely predict similar growth over the next 10 years. That is the good news; the bad news is that this is being achieved through a huge increase in complexity in the hardware, and this

Royal Society Publishing

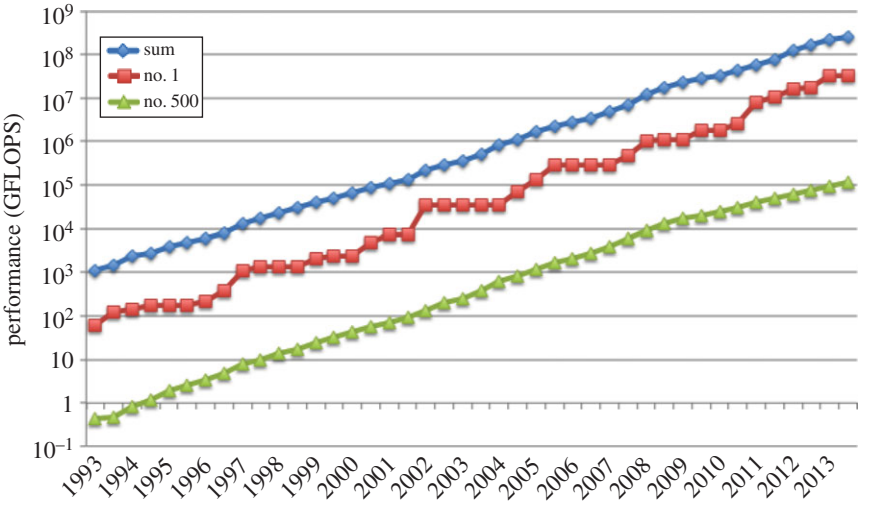

Figure 1. Data showing performance of Top500 supercomputers on the LINPACK test over the past 20 years. The top line is the sum of the top 500 systems, the middle line is no. 1, and the bottom line is no. 500 [1]. (Online version in colour.)

is inevitably complicating the software challenge of exploiting the hardware and trying to achieve a good fraction of its peak potential.

The purpose of this article is to document the changes which have taken place in processor and system design in recent years, and outline some of the likely developments in the next 10 years. This then leads into a discussion of how software is evolving, the challenges which are being faced, and some of the new directions being explored to address them. It concludes with a discussion of the implications for researchers developing new computational fluid dynamics (CFD) codes; what should they think about in designing their new codes, and how can they avoid being locked into any one particular architecture.

An applications person might question why articles like this usually proceed in this sequence: hardware, software, application. In an ideal world, all three might be integrated, and indeed one of the current buzz-words in HPC is 'co-design' which aims to do this for certain critical applications, particularly in the USA [3]. However, the commercial reality is that HPC sales are small compared with the rest of the computer industry. Increasingly, processor development is being driven by the needs of smartphones, tablets, car entertainment/navigation systems and other mobile applications. This is one of the reasons why energy efficiency is a critical concern, although it is also the case that the electricity cost for a major supercomputer can be comparable to its purchase price when spread over its useful lifetime. For example, considering just a single 100 W CPU, if electricity costs 10 p kWh^{-1} then the 3 year electricity cost is approximately £250, compared with a CPU purchase price of £200–1000, depending upon the model. It is also interesting to look at the history of power consumption of the top supercomputers over the past 30 years:

— in 1977, the Cray-1 used 115 kW,
— in 1985, the Cray-2 used 150–200 kW,
— in 1993–96, the Japanese Numerical Wind Tunnel used 500 kW,
— in 2005, the top-ranked IBM Blue Gene/L used 716 kW, and
— today's no. 1, the Chinese Tianhe-2 Xeon Phi system, uses 17 MW.

This growth in power consumption is not sustainable.[1]

[1]On a related note, although there have been comments in the press about the CO_2 emissions of the vast server farms employed by the banks, the CO_2 emissions of the systems used in computational engineering are generally much smaller, and certainly very much less than the CO_2 savings due to improvements in efficiency resulting from computational design.

Hence, to a large extent the hardware does come first in practice, and then the software is designed to achieve the best possible performance on that hardware. If small changes to the hardware can produce major improvements for HPC applications, then those will often be implemented, but the processors and memory subsystem are not designed primarily for HPC applications. Only the highest performing networking is custom-designed for HPC.

In reading the article, the key themes to note are:

— energy efficiency has become very important; this is what has led to the rise of multicore and manycore processors;
— the cost, in both energy and time, of moving data is greater than the cost of performing computations on it; this has major implications for processor/system design and will become a major concern to most application developers; and
— applications must have a very high degree of natural parallelism to exploit all of the processing cores.

2. Current hardware

(a) Memory

It may seem odd to start with the memory subsystem, but this choice emphasizes its central importance. The multicore CPUs and manycore GPUs (graphics processors with thousands of cores) which we will discuss next are each, in different ways, trying to cope with the limitations of the memory subsystem.

Advances in memory technology have struggled to keep pace with the phenomenal advances in processors. This difficulty in improving the main memory bandwidth led to the development of a cache hierarchy with data being held in different cache levels within the processor. The idea is that instead of fetching the required data multiple times from the main memory, it is instead brought into the cache once and re-used multiple times. However, caches are much smaller than the main memory, so there is a limit to how much data can be held within them.

A critical processor design decision is how much of the chip to use for computation, and how much to allocate to cache. In its Xeon server chips which are typically used for HPC applications, Intel allocates about half of the chip to cache, with the largest LLC (last-level cache) being 30 MB in size. IBM's new Power8 CPU has an even larger L3 cache of up to 96 MB [4]. By contrast, the largest L2 cache in NVIDIA's GPUs is only 1.5 MB. On the other hand, the Xeon bandwidth is only up to $60\,\mathrm{GB\,s^{-1}}$ compared to IBM's Power8 bandwidth of up to $192\,\mathrm{GB\,s^{-1}}$, and NVIDIA's latest GPUs with a bandwidth of over $280\,\mathrm{GB\,s^{-1}}$. These different hardware design choices are motivated by careful consideration of the range of applications being run by typical users.

As well as bandwidth, which determines the rate at which large amounts of data are transferred, another important aspect of the memory subsystem is the latency, the minimum amount of time it takes to obtain a single piece of data from the main memory. In both CPUs and GPUs, this is measured in hundreds of clock cycles, so a key aspect of processor design to be discussed later is how the processor copes with this delay by finding other useful work to be performed while one particular instruction is waiting for data.

One complication which has become more common and more important in the past few years is non-uniform memory access. Ten years ago, most shared-memory multiprocessors would have several CPUs sharing a memory bus to access a single main memory. Now, it is standard for memory to be linked directly to a particular processor chip, so in a system with two sockets there is the problem that a process on one chip may need to access data held in the memory connected to the other chip. In this case, there are extra delays involved, and for best performance an application developer may need to avoid this by using a separate MPI process for each chip with local memory allocation.

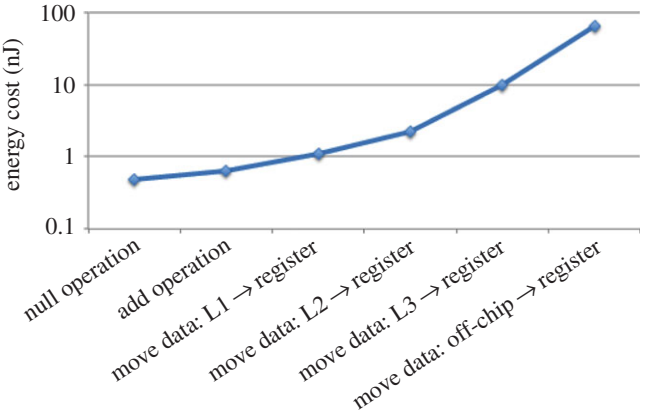

Figure 2. Energy costs for various operations and data movement to registers from different levels in the memory hierarchy, based on [5]. (Online version in colour.)

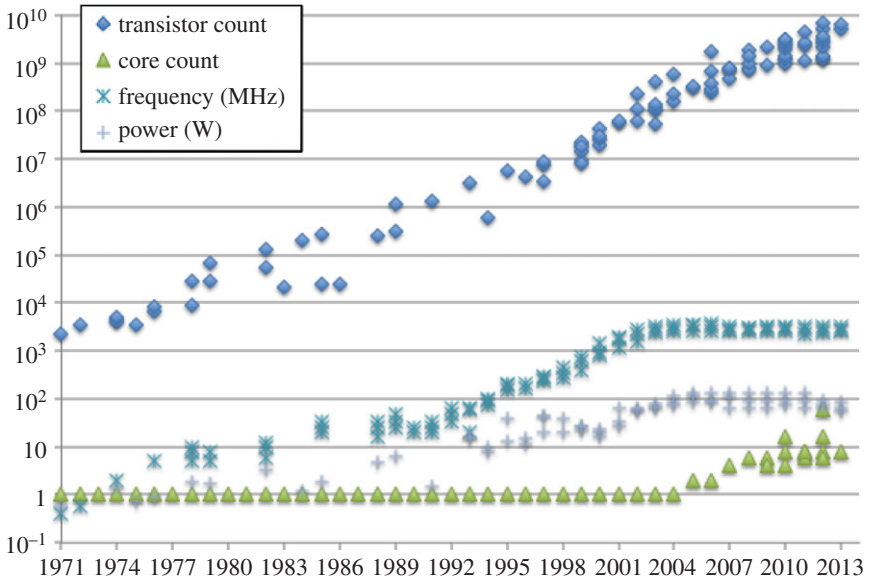

Figure 3. Evolution in transistor count, clock frequency, number of cores and power consumption for processors over the past 40 years. (Online version in colour.)

A final comment on the memory subsystem concerns the energy cost of moving data compared to performing a single floating point computation. As shown in figure 2 from [5] the data movement cost is much greater. This is likely to become increasingly important in the future, with a need to keep data close to where they are being used, as much as possible. In fact, a recent study shows that 28–40% of the total energy consumption is spent moving data, and 19–36% of it is wasted in stalled cycles [5].

(b) Processors

Figure 3 illustrates various aspects of processor evolution over the past 40 years, presenting data obtained from various sources. Perhaps the most important curve is the growth in clock frequency. For many years, CPUs had a single processing core, and the increase in performance came partly

from an increase in the number of computational pipelines, but mainly through an increase in clock frequency. Unfortunately, the power consumption is approximately proportional to the cube of the frequency and this led to CPUs with a power consumption of up to 250 W.

This was not sustainable, and over the past few years the move has been to improve performance through multiple cores running at slightly lower clock frequencies. The increase in the number of cores is made possible by the continuing growth in the number of transistors, which is usually referred to as Moore's law. This is a consequence of shrinking chip feature sizes, with the latest CPUs being manufactured using 22 nm process technology.

CPUs address memory bandwidth limitations by devoting half or more of the chip to LLC, so that small applications can be held entirely within the cache. They address the 200-cycle latency issue by using very complex cores which are capable of out-of-order execution, reordering the execution of instructions within a single thread depending on which instructions have the required input data from main memory or prior instructions. This maximizes single-thread performance which is critical for many mainstream applications. Hence, a very small fraction of the chip is actually devoted to floating point multiplication or addition; most is either for cache or complex management.

By contrast, GPUs adopt a very different design philosophy because of the different needs of the graphical applications they target. A GPU usually has a number of functional units. Each one has a number of very simple cores, using in-order execution, operating in groups effectively as a vector processor. In the particular case of NVIDIA's latest Kepler generation GPUs, there are up to 15 SMX units, and each SMX has 192 cores, arranged in groups of 32 so that the effective vector length is 32 for both single and double precision computations.

GPUs usually have high bandwidth, up to 288 GB s^{-1}, to fast graphics memory. However, the memory latency is still hundreds of clock cycles. To cope with this, GPUs rely on multithreading, with up to 12 threads per core. Each of these threads has its own registers, and so the instruction scheduler can switch very rapidly from one thread to another, depending on which has the input data required to perform its next computation. Note that this implies that the total number of active threads per GPU can exceed 30 000; this means that an application has to have this degree of inherent parallelism to fully exploit a GPU.

The simplicity of the cores and their vector operation, sharing the control logic to decide which instructions to perform next, is what makes it possible to have up to 2880 cores in a single chip, delivering up to 4.3 TFlops in single precision, or 1.4 TFlops in double precision.

CPUs have also introduced vector units. The latest generation Intel CPUs have 256-bit AVX vector units which perform computations on vectors of eight single precision variables or four double precision variables. These are targeted at some of the same multimedia applications as GPUs, such as video/audio streaming, graphics applications, voice recognition. The vector length is likely to double in future designs, and so this vector capability will become increasingly important for maximizing the performance of engineering applications.

There is insufficient space in this article to discuss in detail the Intel Xeon Phi accelerator, AMD CPUs/GPUs or the IBM Blue Gene system based on PowerPC processors. These are all interesting, but we have chosen to focus on Intel CPUs and NVIDIA GPUs as the main alternatives in HPC, now that IBM has chosen to halt further development of the Blue Gene family.

(c) Networking

HPC also relies on high-performance networking, and its importance is perhaps indicated by Intel's 2012 purchase of both Cray's interconnect technology and QLogic's Infiniband (IB) networking business.

Cray's custom networking in its supercomputers achieves 10 GB s^{-1} bandwidth to/from each compute node, with 1–2 μs latency [6], whereas IB networking offers a maximum of 7 GB s^{-1} with similar latency [7]. The IB bandwidth is less than the 8–12 GB s^{-1} bandwidth of the PCIe bus which transfers data between a CPU and a GPU. Hence, in multi-GPU computations the PCIe transfer will not be the bottleneck in performance.

One other note is that IB switches are getting smarter. To reduce the delays in MPI collective communications, in particular reductions (such as computing a maximum over all computing nodes) this processing is now done within the IB switch to avoid the delays involved in sending the data to nodes for processing [8].

3. Future developments

There are a number of interesting developments which are due to emerge from the corporate research laboratories into production hardware in the next few years.

(a) Stacked memory

One of the limitations in improving memory bandwidth is the number of connecting pins required. With stacked memory, a substantial amount of memory (up to 16 GB) is stacked on top of the processor, and connections are drilled down into the processor. Both Intel [9] and NVIDIA [10] are planning to release high-end processors with stacked memory by 2016, with memory bandwidth of up to 1 TB s^{-1}. This will be a significant improvement, but with processors continuing to improve their performance it does not solve the problem of the memory bandwidth bottleneck, and instead adds one more level into the memory hierarchy. It is not yet clear whether this additional level will need to be explicitly managed by program developers, transferring large chunks of data to/from the main memory, or whether it will be handled automatically through caching.

(b) Integrated chips and power management

We already have hybrid chips from Intel, AMD and NVIDIA which combine a CPU with a GPU in a single chip, and in some cases networking is also included in the package. Currently, these are for low-end mobile platforms where there are significant cost and energy savings to be achieved by moving to SoC (system-on-a-chip) designs with just one chip plus external memory.

In the near future, the same development is likely to take place at the high end. Intel has already announced that its next Xeon Phi processor will not need a standard Xeon as a front-end processor, and IBM and NVIDIA have announced plans to integrate IBM's Power8 CPUs and NVIDIA GPUs in future HPC systems [4,9]. Networking is also likely to be integrated onto the high-end chips, with Intel already announcing that Xeon and Xeon Phi chips will appear very soon with integrated Gigabit Ethernet [11].

One advantage of integrating more and more functionality onto a single chip is that it gives more fine-grain control over power management. If one part of the chip, such as the networking, is not being used heavily then it can be turned off, or at least run at a lower frequency to save power and keep the chip within thermal limits. This approach which is sometimes referred to as 'dark silicon' (using only part of a chip due to a limited power budget) is likely to become more common in the near future [12].

This kind of fine-grain power control is already possible; in a multicore Intel CPU, if only one of the cores is active its frequency can be 'boosted' to improve performance. At present, this is all controlled by the hardware itself, but in the future, it is possible that the programmer may be given the opportunity to 'advise' the hardware on the optimal allocation of the power budget between the different components in the chip.

(c) Rise of low-power processors

There is considerable uncertainty about the future of HPC systems. One vision is of 'fat' compute nodes based on processors similar to the Xeon Phi or a high-end GPU/CPU hybrid, but another vision is of a very large number of low-end energy efficient compute nodes.

Intel has an Avoton SoC [13] with 8 cores, 25 GB s^{-1} memory bandwidth, integrated Gigabit Ethernet networking, running at about 10 W. Similarly, NVIDIA has just announced a Tegra K1 chip [14] consisting of a 192-core GPU plus a quad-core 64-bit ARM CPU (but no integrated networking) running at 5 W. It is possible that chips like this, each with a modest amount of (stacked) memory represent the future of HPC, because they may offer the best energy efficiency, as well as leveraging the massive design effort going into energy efficient low-end processors for mobile devices.

(d) Photonics

The previous items are certain to emerge in the next few years. By contrast, photonics is still being developed in the research laboratories; it is not yet clear when it will emerge into commercial products although Fujitsu has demonstrated a server with components connected through optical PCIe using Intel silicon photonics chips [15].

Photonics represents the next stage in integration. Current network cards often use transceivers to convert electrical signals into light to enable networking across fibre-optic cables. In the future, this will be integrated onto the processor, so that chips can be connected optically [16]. This should reduce the latency and power consumption and improve the bandwidth which will be required in the future, particularly if HPC moves towards very large numbers of small energy-efficient processors. At the very least, it should help to keep the networking side of the system evolving in line with the processors.

(e) Requirement for fault tolerance

One concern for the future is that the hardware may become increasingly unreliable. This is partly a concern about an increasing number of components leading to a higher rate of failure, but this is offset by the increasing degree of SoC integration so that the total number of chips is not increasing as rapidly as it might otherwise. The other aspect is that higher energy efficiency is driving chips towards lower operating voltages, and this leads to an increase in radiation-induced 'soft' errors [17,18].

This could be an enormous problem for software developers, so large that the industry may be reluctant to go too far in reducing operating voltages. It is actually a much larger problem for 'everyday' computing than for HPC applications. Most HPC applications already use checkpointing to enable an application to restart from an intermediate point in the event of the failure of a compute node. At present, most checkpointing is performed to an external disk array, and this is becoming more and more time-consuming. However, flash memory is now replacing local disks within compute nodes, and so in the future checkpointing will be done very quickly and easily to local flash memory. To guard against 'hard' errors which bring down a whole compute node, it would be necessary to checkpoint data to a neighbouring compute node, which would take just a little longer.

Another problem identified in [17,18] is that rebooting extremely large systems takes a long time. There is also concern that the operating system software itself is so complex that an increasing proportion of failures are due to software rather than hardware. These both imply a need to develop simpler, more reliable operating systems, at least for extreme-scale systems, which can be rebooted in parallel without significant sequential bottlenecks.

4. Software

Hardware evolves very rapidly, but software evolves on a time scale often measured in decades rather than years.

(a) Languages

The dominant programming languages for HPC computing remain Fortran, C and C++. Each has evolved to some degree over the years, but continuity is essential for big application codes whose lifetime is also measured in decades.

It is extremely difficult for any new language to get established and achieve critical mass in its adoption. Therefore, for better or worse, it seems likely that these will remain the dominant programming languages for the foreseeable future. However, these languages were designed for a different era with a Von Neumann sequential execution model. They were not designed to capture the specification of parallel algorithms, so the compilers must analyse the code to try to extract the inherent parallelism. Hence, these languages will remain very badly suited for the future of massively parallel computing.

(b) Distributed-memory computing

In the domain of distributed-memory computing, MPI message-passing remains dominant, much to the surprise of its original developers. It is still very low level in its nature, but in practice most software developers write their own MPI communication library to perform the tasks required by their application, and thereby achieve good separation between the main application code and the MPI message-passing.

MPI itself evolves slowly. The most significant changes are the inclusion of one-sided communications, and the future addition of asynchronous collective operations (such as finding a sum or maximum across all processes) to avoid the latency of synchronous reductions.

MPI also supports parallel file I/O which is increasingly important to avoid file reading/writing becoming a major application bottleneck. File format standards such as HDF5 are then layered on top of MPI I/O to enable easy access to these facilities.

The only significant challenge to MPI has been the PGAS [19] (partitioned global address space) languages: Chapel [20], Fortress [21] and X10 [22]. However, despite significant levels of US funding for their development, none of them has achieved a significant level of use within the HPC community.

(c) Shared-memory computing

OpenMP remains the dominant standard in shared-memory computing, and there is no reason to think that this will change. OpenMP works through the programmer's insertion of 'pragmas' into the code to specify which loops are to be executed by multiple threads. The latest version, OpenMP v. 4.0 [23], has added support for vectorization through SIMD directives, and this may become the best way to exploit the AVX vector units in Intel CPUs. It has also added support for the offload of computation onto accelerators such as the Xeon Phi.

A similar pragma approach is used in a new open standard called OpenACC to specify which loops are to be off-loaded for execution onto GPU or Xeon Phi accelerators. The movement of data between the CPU and the accelerator can be handled automatically by an OpenACC compiler, or can be controlled by additional pragmas for improved performance. It will be interesting to see whether OpenACC eventually merges into a future version of OpenMP.

(d) CUDA and OpenCL

For programming on GPUs, CUDA is the de facto standard in HPC, even though it is NVIDIA's proprietary extension of C. Its success is due to NVIDIA's GPUs being the dominant accelerator platform in HPC, and the extensive ecosystem (libraries, debuggers, profilers, IDEs, support for Fortran) which has been built around CUDA.

The alternative is OpenCL which is an open standard supported by a number of manufacturers. OpenCL has been widely adopted for mobile devices (phones, tablets, etc.) where

the application portability is important. However, there is no sign yet of it eroding CUDA's dominance in HPC.

(e) AVX vectorization

As the AVX vector length in Intel CPUs increases, it will become increasingly important for HPC applications to achieve good AVX vectorization; otherwise as much as 80% of the compute capability might be wasted [24], although this comment depends on whether the execution time is dominated by the computation or the data movement.

Currently, it can be difficult to achieve vectorization, and Intel offers a number of competing approaches. At the lowest level, vector intrinsics are the most reliable way to maximize performance, but the programming is rather tedious. At the highest level, one can rely on auto-vectorization by the Intel compiler, but it can usually only handle very simple codes. In between, Intel offers a proprietary extension to C/C++ called Cilk Plus [25], as well as both OpenMP and OpenCL. Hopefully, this collection will be whittled down to a single effective and easy approach in the future.

(f) Libraries

One response to the increasing complexity of modern architectures is for application developers to rely heavily on high-performance libraries. It is then the library developers' responsibility to port the libraries onto new hardware platforms. This has been quite successful with accelerators, with many standard libraries ported to CUDA, and an increasing number also ported to OpenCL and/or the Xeon Phi.

For example, MAGMA [26], the accelerator version of the LINPACK library for dense linear algebra, has both single and multiple GPU support, has almost the full LINPACK coverage in CUDA and also has versions of many routines for both OpenCL and the Xeon Phi. One interesting aspect of MAGMA is the dynamic execution of task DAGs (directed acyclic graphs). A linear algebra task is decomposed into a set of smaller block tasks, and a run-time system tracks the interdependencies, launching each task when it has the necessary inputs. This style of programming and execution may become much more common in the future.

5. Software research

The challenges of manycore computing have spawned a number of different lines of research, and here we choose to discuss three of them.

(a) Domain-specific languages and high-level abstractions

We have already said that Fortran and C/C++ are likely to remain the dominant general-purpose languages for scientific computing, despite the fact that they are fundamentally not designed to capture the specification of parallel algorithms.

There is an alternative: the development of domain-specific languages (DSLs) or other high-level abstractions (HLAs). The aim here is not to produce a new general-purpose language, but instead design a DSL or HLA which will address the needs of developers in a particular application domain, such as CFD, through enabling a high-level specification of their algorithms and data dependencies. By using knowledge of the application domain, it is possible to better capture the inherent parallelism in the algorithms, and generate code for efficient execution. Furthermore, as the underlying hardware changes, the user's code does not need to change; only the DSL or HLA has to change to target new hardware platforms.

Examples in the area of CFD are SBLOCK [27] for structured grids, OP2 [28,29] and Liszt [30] for general unstructured grid computations, and FEniCS for finite-element computations [31]. SBLOCK uses an algorithm specification expressed in Python to generate the required code for

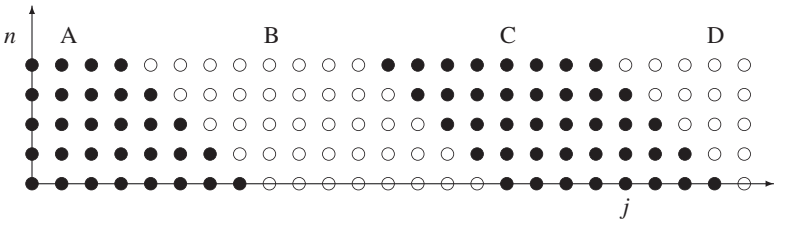

Figure 4. An illustration of tiling to reduce data communications. All of stripe A is executed first, then B, C and D.

different platforms. OP2 uses source code transformation, converting regular Fortran or C++ code to the required backend code, such as CUDA for NVIDIA GPUs. Liszt is a DSL based on a language called Scala which is similar to C++. While these first three provide support for general algorithms on their respective grid types, allowing the user to choose the numerical algorithm, FEniCS operates at an even higher level, with the user specifying the partial differential equation to be solved, and then FEniCS offers a selection of finite-element discretizations to be used.

Although this is a very promising direction for research, there are two main difficulties. One is designing a DSL or HLA which is sufficiently general to handle a number of different applications, but small enough to be implementable by a single research group. The other is the challenge of securing funding to maintain the DSL or HLA so that users can rely on it in the future.

(b) Auto-tuning

Optimizing application codes for new architectures is often a process of trial-and-error. Even experts find it hard to make the optimal choices, faced with decisions such as the number of threads to use, the placement of variables, and the tradeoff between storing and reusing data versus re-computing on-the-fly. Furthermore, the optimal choice may change from one generation of hardware to the next.

Increasingly, the solution to this is to use auto-tuning, programming multiple implementations with parameters controlling different aspects of the execution [32]. The optimal values for those parameters can then be determined computationally using auto-tuning software. Most auto-tuners optimize a representative short-running testcase and then rely on real applications behaving similarly, but there is also a line of research on real-time auto-tuning for long-running applications.

(c) Communication avoiding algorithms

Earlier we discussed the fact that the execution cost of an algorithm (both in time and energy) increasingly depends on data movement, and not the number of floating point operations. This has led to research to develop ways of reducing the amount of data moved [33]. Such research is not new, since 'blocking' has been used for many years to optimize caching in dense matrix operations, but it is now increasing in importance.

One aspect of this which may become important for CFD calculations, especially on structured grids, is 'tiling'. This is illustrated in figure 4 for a one-dimensional application with explicit time-marching so that u_j^{n+1} depends on $u_{j-1}^n, u_j^n, u_{j+1}^n$. Normally, one would compute all of time level n before proceeding to time level $n+1$, but with tiling one can compute in 'stripes' as indicated, completing all of stripe A before going on to stripe B, then stripes C and D. The size of each of the stripes is chosen so that the data can remain resident within the LLC. Thus, the data are loaded once, and then several time steps take place before it is written back out to the main memory [34,35]. In simple cases, this can be implemented easily, but in real-world applications it becomes very complex. However, memory bandwidth is likely to be the primary constraint on high performance in the future and so such techniques may become essential.

6. Advice for computational fluid dynamics users

Many people in engineering are in the situation of being users of CFD codes, often developed commercially. What are the implications of the hardware and software developments for these users?

One consequence is that the commercial CFD companies are almost certainly struggling to cope with the increased complexity. This may mean that they choose not to adapt their software to run on the most efficient novel architectures. It may also mean that they have less time to spend on adding new capabilities (such as a new turbulence model) to their software, because of the effort they have to devote to hardware porting. It is also likely to pose a huge barrier to new entrants into the commercial CFD software business, unless they have a novel approach which allows them to easily cope with the diversity of modern hardware architectures.

Another consequence for users is that they should think very carefully about the hardware platforms they buy on which to run their CFD codes. In many cases, it is more important to buy hardware with the best memory bandwidth (usually determined by the product of the memory clock frequency and the number of memory channels) rather than being concerned about the compute performance of the CPUs.

To minimize the cost of checkpointing, users should read the analysis in appendix B of [17]. This shows that if the mean time between failures is M and the time taken to create a checkpoint is C, then the optimal checkpointing interval is approximately $\sqrt{2CM}$. This balances the time lost when rewinding to the last checkpoint after a failure, with the time taken to generate the checkpoints. Using this value, the execution efficiency is approximately $1 - \sqrt{2C/M}$. If $M = 1$ day and $C = 1$ min, this gives a checkpointing interval of approximately 1 h, and an efficiency of 96%.

A final comment concerns user expectations. In the past, one was sometimes interested in running big calculations faster than before. Now, the performance of individual cores is increasing very little, so the emphasis with increasing core counts, and increasing number of compute nodes, is either on being able to run much bigger calculations than before, or being able to run many more calculations than before. The latter capability is perhaps particularly useful for users who are increasingly interested in uncertainty quantification.

7. Advice for computational fluid dynamics developers

How does all of the discussion in the previous sections about hardware and software developments relate to the development of CFD codes?

(a) Legacy codes

Existing CFD codes for large-scale applications are likely to have been written as distributed-memory codes using MPI with a single thread for each MPI process. Porting such a code to a novel architecture such as GPUs is likely to be too substantial a task to contemplate; the effort would be similar to writing a brand new code.

The first advice would be to determine whether the existing performance is limited by the compute capability of the CPUs or the memory bandwidth. If it is the former, then attention should perhaps be focused on the vectorization of the code so that it exploits the AVX vector units on modern CPUs. However, it is much more likely to be the latter, in which case attention should be devoted to reducing the amount of data transfer required, even if this introduces additional computations.

For example, in three-dimensional structured grid computations, codes are sometimes written in a style in which each time step involves four passes through the data, one in each coordinate direction to compute a viscous flux through the control volume faces in that direction, and then a final pass to compute the flux residual and update the solution. This approach generates lots of data transfer; it might well be better to use just a single pass through the grid, even though

this might involve computing the viscous fluxes twice, once for each of the cells on either side of the face.

With unstructured grids, the simplest way to improve performance is to renumber nodes, cells and faces to maximize cache efficiency by ensuring, for example, that nodes which are close to each other have similar indices. This ensures that when a cache line is loaded most of the cache line is used, and so the same cache line does not have to be loaded many times. Many grid generators, especially those based on Delauney triangulation ideas, generate grids with a numbering scheme which seems almost random, so renumbering can produce substantial benefits [36].

With increasing hardware/software unreliability, checkpointing is essential. As well as optimizing the checkpointing interval, as described in §6, effort should also be put into minimizing the size of the checkpoint dataset, storing the least possible information required to restart the calculation. Allied to this is the use of parallel file I/O to achieve scalable I/O performance when moving to huge datasets on large distributed-memory systems.

(b) New code development

One obvious point is that applications need a very high degree of parallelism to exploit large numbers of manycore processors, and researchers will need to keep this in mind in developing new numerical algorithms.

Another key point is that CFD codes often have a lifespan of more than 20 years, from initial development, through validation, to productive use and eventual retirement to be replaced by a new code. This lifespan is huge compared with the pace of hardware development in HPC. Codes which are in production use today, such as the Rolls-Royce HYDRA code, were developed at a time when CPUs had only one core, and no-one could imagine that GPUs would become a platform for HPC computations.

Similarly, it is very hard to predict today what is likely to be the most powerful HPC platform in 2030, so anyone developing a new code today has a problem. Ideally, they would like to achieve a high level of performance on current architectures, but they would also like a 'future-proof' capability so that it will also perform well on future architectures.

There are three possible approaches which could be taken. The most conservative is to develop a code to execute well on Intel CPUs, since it is likely that this will deliver good performance for many years. Although the floating point performance of CPUs is not as high as GPUs, they do have very large caches which help to reduce the amount of data transfer to and from the main memory. In that sense, it is a safe choice; it is unlikely the performance will be poor, although it may not be the best possible. In following this approach, it would be good to tackle the challenges of vectorization, since without this an increasing amount of the CPU's capability would be thrown away. However, performance may well be bandwidth-limited, and so vectorization may deliver limited benefits unless one also adopts techniques such as tiling to reduce data transfer, and this requires very significant changes to any code.

The second approach to achieving future-proof performance is to use OpenACC, or any future equivalent within OpenMP. This approach may work well for structured grid applications, leaving the compiler the task of then mapping the required execution onto the target hardware platform, which may be CPUs, GPUs or some other novel architecture. The limitation is that again the application may well be memory-bandwidth limited, so in the long term high performance will require the use of techniques such as tiling to reduce the data transfer to and from the main memory. This is likely to pose a huge challenge to compilers and could be too difficult to address.

The final approach is the use of a DSL or a high-level framework. In this case, the developer specifies what is required at a very high level, and then it is the responsibility of the DSL or framework to implement it as well as possible on a variety of target architectures. Because of the amount of information which can be captured from a high-level specification, the implementation can include features like automatic checkpointing for large distributed-memory computations,

and tiling to reduce data transfer and improve performance for applications which are memory-bandwidth limited.

From the point of view of the application developer, this last approach has the potential to deliver the best performance with the least effort, since the effort is transferred onto the shoulders of the team developing the DSL or framework. However, it is also perhaps the riskiest strategy because the application developer needs to have confidence that this team has (a) the technical expertise to successfully migrate the DSL/framework onto new hardware platforms and (b) the commitment and the funding to carry out (a).

We conclude with two final pieces of advice for application developers. One is to keep an eye on future hardware developments, and think about their implications for the performance of your algorithms and the way in which you implement them. We are in a period of very rapid hardware innovation giving rise to increasing hardware complexity, and unfortunately there is no sign that this will reduce in the next 5–10 years.

The other piece of advice is to get used to the idea that the cost (both in time and energy) of executing an algorithm is as much to do with data movement as floating point calculations. This needs a re-assessment of the algorithms which are used; for example, it might be better in the future to use higher order algorithms which require less memory and data transfer, but an increased number of floating point operations for the same level of accuracy.

Acknowledgements. This paper has been written as part of the EPSRC-funded CCP ASEArch project on Algorithms and Software for Emerging Architectures.

Data accessibility. Further weblinks on hardware and software developments in this area are available from the project website at http://www.oerc.ox.ac.uk/projects/asearch/.

References

1. Top500. 2014 Top 500 supercomputer sites. See http://www.top500.org/.
2. Heroux M, Dongarra J. 2013 *Toward a new metric for ranking high performance computing systems*. Technical report SAND2013-4744. Albuquerque, NM: Sandia National Laboratories. See http://www.sandia.gov/ ~maherou/docs/HPCG-Benchmark.pdf.
3. Advanced Scientific Computing Research (ASCR) Co-Design. See http://science.energy.gov/ascr/research/scidac/co-design/ (accessed 4 February 2014).
4. IBM Power Systems S812L and S822L. See http://www-03.ibm.com/systems/power/hardware/s812l-s822l/specs.html (accessed 14 May 2014).
5. Kestor G, Gioiosa R, Kerbyson D, Hoisie A. 2013 Quantifying the energy cost of data movement in scientific applications. In *IEEE Int. Symp. on Workload Characterization* (*IISWC*), pp. 56–65. (doi:10.1109/IISWC.2013.6704670)
6. Cray XC-30 networking. See http://www.cray.com/Assets/PDF/products/xc/CrayXC30Networking.pdf (accessed 4 February 2014).
7. InfiniBand performance benchmarks. See http://www.mellanox.com/page/performance_infiniband (accessed 4 February 2014).
8. Mellanox Fabric Collective Accelerator. See http://www.mellanox.com/related-docs/prod_acceleration_software/FCA.pdf (accessed 4 February 2014).
9. Intel Knights Landing. See http://en.wikipedia.org/wiki/Xeon_Phi (accessed 4 February 2014).
10. NVIDIA Volta GPUs will have stacked DRAM. See http://blogs.nvidia.com/blog/2013/03/19/nvidia-ceo-updates-nvidias-roadmap/ (accessed 4 February 2014).
11. Intel to pull networking into Xeon and Xeon Phi chips. See http://www.enterprisetech.com/2013/11/25/intel-pull-networking-xeon-xeon-phi-chips/ (accessed 4 February 2014).
12. Esmaeilzadeh H, Blem E, St. Amant R, Sankaralingam K, Burger D. 2011 Dark silicon and the end of multicore scaling. In *Proc. 38th Annual Int. Symp. on Computer Architecture*, *ISCA '11*, pp. 365–376. New York, NY: ACM. (doi:10.1145/2000064.2000108)
13. Intel Avoton chips. See http://ark.intel.com/products/codename/54859/Avoton (accessed 4 February 2014).
14. NVIDIA K1 chips. See http://www.nvidia.co.uk/object/tegra-k1-processor-uk.html (accessed 4 February 2014).

15. Fujitsu lights up PCI Express with Intel silicon photonics. See https://communities.intel.com/community/itpeernetwork/datastack/blog/2013/11/05/fujitsu-lights-up-pci-express-with-intel-silicon-photonics (accessed 4 February 2014).
16. Jalali B, Fathpour S. 2006 Silicon photonics. *J. Lightwave Technol.* **24**, 4600–4615. (doi:10.1109/JLT.2006.885782)
17. Snir M *et al.* 2013 *Addressing failures in exascale computing*. Technical report ANL/MCS-TM-332. Argonne, IL: Argonne National Laboratory.
18. Cappello F, Geist A, Gropp W, Kale L, Kramer W, Snir M. 2009 Toward exascale resilience. *Int. J. High Perform. Comput. Appl.* **23**, 374–388. (doi:10.1177/1094342009347767)
19. Yelick K *et al.* 2007 Productivity and performance using partitioned global address space languages. In *Proc. 2007 Int. Workshop on Parallel Symbolic Computation*, pp. 24–32. New York, NY: ACM. (doi:10.1145/1278177.1278183)
20. Chamberlain B, Callahan D, Zima H. 2007 Parallel programmability and the Chapel language. *Int. J. High Perform. Comput. Appl.* **21**, 291–312. (doi:10.1177/1094342007078442)
21. Allen E, Chase D, Hallett J, Luchangco V, Maessen J-W, Ryu S, Steele GL, Tobin-Hochstadt S. 2008 *The Fortress language specification*. Technical report. Santa Clara, CA: Sun Microsystems, Inc.
22. Charles P, Grothoff C, Saraswat V, Donawa C, Kielstra A, Ebcioglu K, von Praun C, Sarkar V. 2005 X10: an object-oriented approach to non-uniform cluster computing. In *Proc. 20th Annual ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '05*, pp. 519–538. New York, NY: ACM. (doi:10.1145/1094811.1094852)
23. OpenMP. See http://openmp.org/ (accessed 4 February 2014).
24. Intel IA-64 and IA-32 architectures optimization reference manual. See http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf (accessed 30 April 2014).
25. Cilk Plus. See https://www.cilkplus.org/ (accessed 4 February 2014).
26. MAGMA. See http://icl.utk.edu/magma/ (accessed 4 February 2014).
27. Brandvik T, Pullan G. 2010 SBLOCK: a framework for efficient stencil-based PDE solvers on multi-core platforms. In *Proc. 10th IEEE Int. Conf. on Computer and Information Technology*, pp. 1181–1188. Washington, DC: IEEE Computer Society. (doi:10.1109/CIT.2010.214)
28. Giles M, Mudalige G, Sharif Z, Markall G, Kelly P. 2012 Performance analysis and optimization of the OP2 framework on many-core architectures. *Comput. J.* **55**, 168–180. (doi:10.1093/comjnl/bxr062)
29. Mudalige G, Giles M, Reguly I, Bertolli C, Kelly P. 2012 OP2: an active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures. In *Innovative Parallel Computing, San Jose, CA, 13–14 May 2012*. (doi:10.1109/InPar.2012.6339594)
30. DeVito Z *et al.* 2011 Liszt: a domain specific language for building portable mesh-based PDE solvers. In *Proc. 2011 Int. Conf. for High Performance Computing, Networking, Storage and Analysis, SC '11*, pp. 9:1–9:12. New York, NY: ACM. (doi:10.1145/2063384.2063396)
31. Logg A, Mardal K-A, Wells G. 2012 *Automated solution of differential equations by the finite element method*. Berlin, Germany: Springer.
32. Spencer B. Flamingo: parameter auto-tuning for parallel programs. See http://mistymountain.co.uk/flamingo/ (accessed 4 February 2014).
33. Demmel J, Hoemmen M, Mohiyuddin M, Yelick K. 2008 Avoiding communication in sparse matrix computations. In *IEEE Int. Symp. on Parallel and Distributed Processing, IPDPS 2008, Miami, FL, 14–18 April 2008*. (doi:10.1109/IPDPS.2008.4536305)
34. Frigo M, Strumpen V. 2005 Cache oblivious stencil computations. In *Proc. 19th Annual Int. Conf. on Supercomputing, ICS '05*, pp. 361–366. New York, NY: ACM. (doi:10.1145/1088149.1088197)
35. Giles M, Mudalige G, Bertolli C, Kelly P, Laszlo E, Reguly I. 2012 An analytical study of loop tiling for a large-scale unstructured mesh application. In *2012 SC Companion: High Performance Computing, Networking, Storage and Analysis* (*SCC*), pp. 477–482. (doi:10.1109/SC.Companion.2012.68)
36. Burgess D, Giles M. 1997 Renumbering unstructured grids to improve the performance of codes on hierarchical memory machines. *Adv. Eng. Software* **28**, 189–201. (doi:10.1016/S0965-9978(96)00039-7)